SYBEX

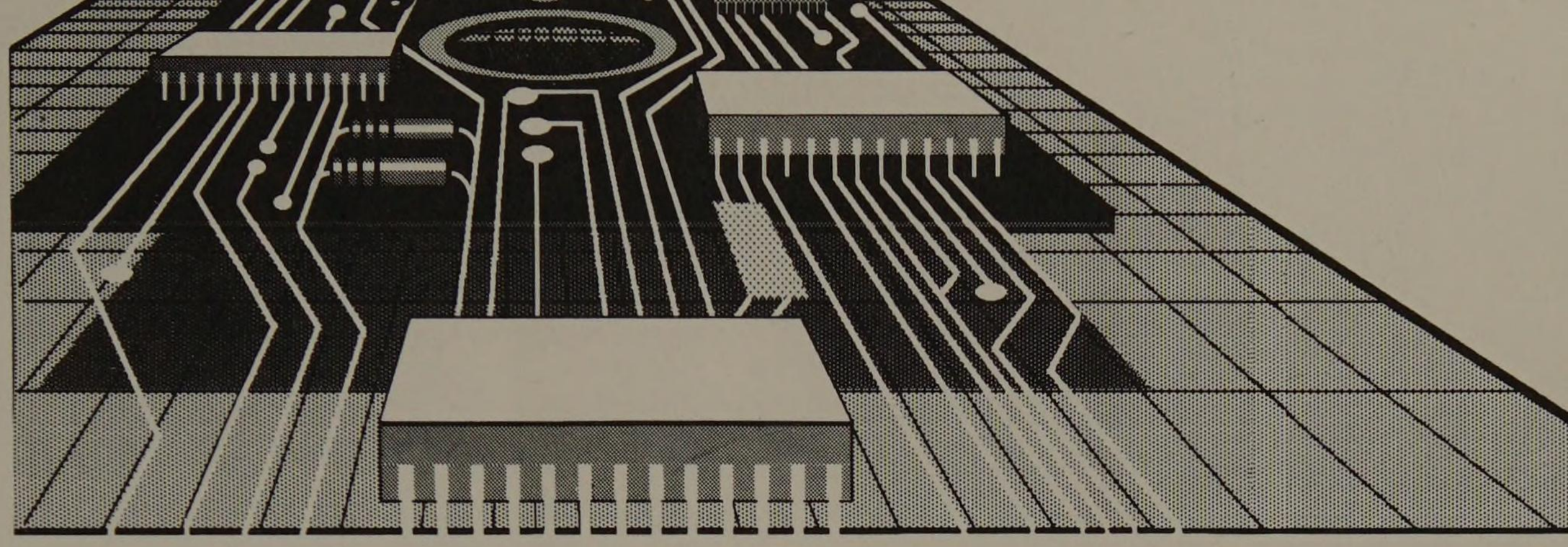
THE MS™-DOS HANDBOOK

Richard Allen King

Second Edition

SYBEX

The MS-DOS Handbook



The MS™-DOS Handbook

Second Edition

Richard Allen King

SYBEX®

San Francisco • Paris • Düsseldorf • London

Cover art by Jean-Francois Pénichoux

Altos is a registered trademark of Altos Computer Systems.
Compaq is a trademark of COMPAQ Computer Corporation.
CP/M and CP/M-86 are registered trademarks of Digital Research, Inc.
Crosstalk is a trademark of Microstuff.
Delphi is a trademark of General Videotex Corporation.
Dow Jones News/Retrieval Service is a service mark of Dow Jones Co.
86-DOS is a registered trademark of Seattle Computer Products, Inc.
IBM, IBM PC, and PC Network are registered trademarks of International Business Machines Corporation.
Intel is a trademark of Intel Corporation.
ITT Xtra is a trademark of ITT Corporation.
Lotus 1-2-3 is a trademark of Lotus Development Corporation.
MS and XENIX are registered trademarks of Microsoft Corporation.
Multiplan is a trademark of Microsoft Corporation.
NorthStar is a trademark of North Star Computers, Inc.
PC-Talk is a trademark of Freeware.
pfs:Access is a trademark of Software Publishing Corporation.
ProKey is a trademark of RoseSoft.
QNX is a trademark of Quantum Software Systems, Inc.
Sidekick is a trademark of Borland International, Inc.
6502 is a registered trademark of MOS Technology.
UNIX is a trademark of Bell Laboratories.
VisiCalc is a registered trademark of VisiCorp, Inc.
WordStar is a registered trademark of MicroPro International, Inc.
Xeno-Copy is a trademark of Vertex Systems, Inc.
SYBEX is a registered trademark of SYBEX, Inc.

SYBEX is not affiliated with any manufacturer.

Every effort has been made to supply complete and accurate information. However, SYBEX assumes no responsibility for its use, nor for any infringements of patents or other rights of third parties which would result.

Copyright©1986 SYBEX Inc., 2021 Challenger Drive #100, Alameda, CA 94501. World rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic or other record, without the prior agreement and written permission of the publisher.

Library of Congress Card Number: 86-61063
ISBN 0-89588-352-X
Manufactured in the United States of America
10 9 8 7 6 5 4

To Jonathan

ACKNOWLEDGMENTS

I would like to thank Dr. Rudolph S. Langer, who suggested the subject of this book, and from whom I learned most of my better programming practices (the others are mine); Charlie Hoy, of Zenith Data Systems, Ron Shafer, of AAO Computer Services, Inc., and Stephen Michel of Computerland in El Cerrito, for assistance in providing access to MS-DOS; Valerie Robbins for her valuable editorial improvements; and my wife Janet for her wholehearted support.

Richard Allen King
May 1986

TABLE OF CONTENTS

Introduction xvii

How to Use this Book xviii
Conventions Used in the Book xviii
The History of MS-DOS xix
Summary of the Chapters xx

PART I: THE PROGRAMMER'S HANDBOOK

1 MS-DOS—A First View 1

Introduction 1
Operating Systems 2
MS-DOS Fundamentals 2
MS-DOS and ROM BIOS 3
Features of the 8086/8088 Family 5
Function Calls in MS-DOS 8
The Components of MS-DOS 15
Summary 17

2 Files in MS-DOS 19

Introduction 19

An Outline of Files in MS-DOS 20
Locating Files in MS-DOS 20
File Attributes and File Access Modes 22
Using File-Access Function Calls 23
Whole File Access 24
Reading and Writing Data Using File Handles 27
Trading File Handles 29
Finding Files in a Directory 30
Making, Removing, and Changing Directories 30
Utility Functions 31
Version 1 File Handling and Data Access 33
The Structure of an MS-DOS File—Version 1 34
File Control Blocks 34
Accessing a File and Its Data in Version 1 37
Working with a Single File 38
Finding, Creating, and Deleting Files—Version 1 44
FCB-Related Utility Functions 48
Summary 49

3 Disks in MS-DOS 51

Introduction 51
Disk Sector Layout in MS-DOS 52
Disk Sector Numbering 54
Disk Maps, or What's Where on an MS-DOS Disk 55
Disk Capacities 56
Allocation of Disk Space to an MS-DOS File 58
Disk Access Mechanisms 59
Programming for Nonstandard Disk Drives 61
Summary 63

4 MS-DOS and the Keyboard 65

Introduction 65
Break-Character Processing 66
Keyboard Status Sampling 67

To Echo or Not to Echo 67
To Wait or Not to Wait 68
Buffered Input of Character Strings 69
Dumping Unwanted Keystrokes 69
Changing the Meaning of the Keys 70
Single-Character Input Functions 78
Summary 79

5 Output to the Display Monitors 83

Introduction 83
Function Calls That Affect the Screen 84
Character Attributes 85
Extended Screen Control 89
Video I/O Using the ROM BIOS Function Calls 93
Locations of the Characters and Their Attribute Bytes 100
Summary 100

6 The Serial and Parallel Ports 103

Introduction 103
MS-DOS Serial Port Function Calls 104
ROM BIOS Serial I/O Function Calls [PCCs] 105
The Hazards of Programming for RS-232 Communications 108
The Parallel Printer Interface 109
Summary 110

7 Dates and Times in MS-DOS 113

Introduction 113
Date Formats in MS-DOS 114
Time Formats in MS-DOS 115
ROM BIOS Time 118
Total Time Control 120
Summary 121

8 Error-Handling and Memory-Resident Programs 123

Introduction 123
Memory Management Functions 124
The Execute Function 125
Function Call Error Reporting 130
Summary 135

9 Networking with MS-DOS 137

Introduction 137
Network Concepts 138
MS-DOS Functions under Network Operations 139
Network-Specific Functions in MS-DOS 142
Summary 145

PART II: THE USER'S HANDBOOK

10 MS-DOS—A User's First View 149

Introduction 149
A Brief Description of MS-DOS Capacities 150
MS-DOS and the Keyboard 151
MS-DOS and Disk Errors 154
What's Where in the DOS Manual 155
Print Spooling with the PRINT Command 157
Graphics to the Printer 158
Pipes, Filters, and I/O Redirection 158
Summary 161

11 A Quick Look at EDLIN 163

Introduction 163
Line Numbers in EDLIN 164

The EDLIN Commands 165
Creating a File Using EDLIN 166
The List Command 167
The Page Command 167
The Insert Command 168
The Edit Command 168
The Search and Replace Commands 170
The Append and Write Commands 171
The Transfer Command 171
The Copy Command 172
The Move Command 173
Editing Binary Files 173
Ending an Editing Session 173
EDLIN's Treatment of .$$$ and .BAK Files 174
Summary 174

12 Your MS-DOS Files 177

Introduction 177
Files in MS-DOS 178
Summary 194

13 Configuring the System 197

Introduction 197
Configuration Parameters and the Configuration File 198
Configuration Commands 201
Hard Disks and File Management 206
Assorted Useful Commands 209
Using MS-DOS in a Network 213
Summary 214

14 Batch Operations 217

Introduction 217
Applications for Batch Files 218

How Batch Files Work 218
Creating a Batch File 220
FOR, IF, GOTO, SHIFT, and ECHO 223
A Note or Two about Batch Files 229
Summary 230

15 The Monochrome and Color/Graphics Screens 233

Introduction 233
The Monochrome Display 235
The Color and Graphics Capabilities 235
Color Selection with the Color/Graphics Adaptor 237
A Small Puzzle for Software Writers 241
Choosing a Monitor 241
Summary 242

16 Using the Serial Port 245

Introduction 245
MS-DOS Communications Capabilities 246
Using a Printer with a Serial Interface 246
Potential Problems When Using the Serial Port 247
Connection with Information Retrieval Services 249
Summary 250

17 The Delights of DEBUG 253

Introduction 253
What DEBUG Can Do for You 254
How DEBUG Works 255
How to Use DEBUG 256
Patching with DEBUG 261
Creating Programs with DEBUG 263
Exploring with DEBUG 266
DEBUG Command Summary 269
Summary 270

APPENDICES

A MS-DOS, PC-DOS, and Compatibility 273

B The Different MS-DOS Version Numbers 283

C Function Call Summaries 295

D Interrupt Vectors 305

E Memory Maps 309

F Address Conversion Tables 315

G MS-DOS Command Summary 319

Index 329

INTRODUCTION

This book is written for people who know a little about computers. It assumes that you are using a computer that runs the MS-DOS operating system, and that you need a quick way of knowing more about some of the tools and features you have to work with.

If you intend to write programs that will run within MS-DOS (and PC-DOS), the first part of the book, Chapters 1 through 9, is for you. It shows you the bare necessities of programming the Intel 8086 family of microprocessors that are the heart of MS-DOS computers. Then it shows you the functions your program can call within MS-DOS, along with the structures (files and programs) and devices (keyboard, screens, disks, and serial and parallel ports) on which the functions operate. Reading the first half of the book will give programmers, both first-time and experienced, the background to proceed with confidence.

If you want to become adept at using MS-DOS at the command level, the second part of the book, Chapters 10 through 17, gives you quick access to the features and capabilities of the operating system. For instance, in the section on using the keyboard in Chapter 10, you will see how to use a single key to recall the last command, instead of having to retype it. The user-oriented half of the book also shows how to use batch operations, how to set up a filing system, what you need to know about character output to the screen, how to set up your hard disk, how to configure your system for optimum performance, and a variety of other useful information.

You can skip the whole technical/programming (first) half of the book with no qualms at all, if that suits your purpose. After all, you can drive a car all over North America without knowing much at all about what is under the hood. The second part, Chapters 10 through 17, covers a lot of what you need to know to make MS-DOS work for you.

This book is written to save you time searching around in the manuals that come with your MS-DOS system, and to provide information that cannot be found there. Key features described not at all, or in obscure places, are brought to light here and placed in their logical context. Nonetheless, it is assumed that you have available the primary MS-DOS user's guide. If you also have available more technical manuals, so much the better. They will be useful for the more complex programming tasks you might someday undertake, although information needed from the technical manuals is usually supplied here. This book is not meant to replace the MS-DOS manual; it is an extension and clarification of it. For instance, the basic DOS commands, ASSIGN through VOL, are usually described well in the MS-DOS manuals, so they are not described in detail here. Instead, the MS-DOS Command Summary, with notes on each command as appropriate, is provided in Appendix G.

HOW TO USE THIS BOOK

If you want to find out all you can about a subject, read the chapter with the relevant title and look up all appropriate index entries. This type of organization allows the author to avoid the repetition that would occur if it were necessary to exhaustively include everything on a subject in one central location.

Once you have read this introduction and Chapter 1, feel free to move on to any chapter that interests you. The book has independent chapters on each subject. It need not be read from beginning to end, as if it were a tutorial.

CONVENTIONS USED IN THE BOOK

All numbers are expressed in decimal (base 10) unless they are followed by an H (e.g., 21H), in which case they are hexadecimal (base 16). Some laxity occurs in expressing single-digit numbers less than ten, since the base is not relevant there; also, numbers that are obviously hexadecimal (numbers that contain letters) are not necessarily followed by an H.

The letter K, when referring to memory sizes and disk capacities, indicates the number 1024. So 64K, a convenient way of referring to the space addressable by 16 bits, means 65,536.

The word *disk* refers to both diskettes and the hard disk. Where information relevant only to the hard disk is presented, *hard disk* is used. A *rigid* disk refers to the 3½-inch diskette, as opposed to the *floppy* 5¼-inch diskette.

The name of a file in MS-DOS terminology consists of a filename of one to eight characters, or a filename (of one to eight characters) followed immediately by a period and a filename extension of one to three characters. In this book the term *file name* (note the space) refers to the whole name of the file, which is either *filename* without an extension, or *filename.extension*. This distinction may seem pedantic, but it is useful later on.

Version 1 refers to all the releases of version 1, including 1.0, 1.1, and 1.25. Likewise, *version 2* refers to versions 2.x, including 2.0, 2.02, 2.1, 2.11, etc. *Version 3* refers to versions 3.x.

The *8086/8088 family* of microcomputers now includes advanced versions of these microcomputers. For example, Intel's 80286 and 80386 will run MS-DOS, and NEC makes the V20 and V30 microcomputers that can replace the 8088 and 8086. A multitude of computers now offer the MS-DOS operating system. This book is about generic MS-DOS. However, the majority of users working with MS-DOS use it on a computer that is compatible with one of the IBM Personal Computers (PC, XT, PC*jr,* or AT). In consequence, more than half of the examples in this book are taken from the world of the IBM PC compatibles, and more depth of information is provided for these machines. Where information is specific to the IBM PC compatibles, you will see the notation [PCCs]. For more information on the difference between MS-DOS and PC-DOS, and what constitutes an IBM PC-compatible machine, see Appendix A.

The words "DOS manual" in this book refer to the document(s) supplied by your computer manufacturer that describe(s) all the MS-DOS commands. It might be called a User's Guide or a Reference Guide, or just DOS Manual.

We are about done with the introductory section, save to tell about the history and origins of MS-DOS, and then summarize each of the chapters.

THE HISTORY OF MS-DOS

In 1980, Seattle Computer Products needed an operating system to support their board-level 8086 microprocessor for S-100 systems. The ancestor of MS-DOS was born that summer, and they named it QDOS. As happens with most software, the MS-DOS we now know and love has come a long way from the original operating system, although it owes its basic structure to that dim past progenitor.

Tim Paterson is the programmer who wrote QDOS. Being a programmer of sensitivity and humor, he called it QDOS because that's what it was—a quick and dirty operating system. Quick and dirty describe the origins of a few fine (and many more not-so-fine) products. By the end of 1980, QDOS had become 86-DOS, not only because programmers' humor is not

favored by those who sell, but also because 86-DOS was significantly better than QDOS and deserved another name.

Microsoft had earlier bought rights to the 86-DOS software. When IBM decided to venture into the microcomputer business, it chose Microsoft as one of the possible developers of an operating system for its line of personal computers. Within shrouds of secrecy, a team of Microsoft's programmers, using 86-DOS as a base, developed the now-revealed MS-DOS on a prototype IBM Personal Computer. By May 1981 that team included Tim Paterson. Just before the release of the IBM PC, Microsoft closed its grip on 86-DOS by purchasing exclusive rights to it from Seattle Computer, renaming it MS-DOS at the same time. MS-DOS is now available on a multitude of 8086/8088-based personal computers.

Now that you know the origins of MS-DOS, please do not leap to the phone to call Microsoft with your queries and suggestions. Each manufacturer of the computer is responsible for the version(s) of MS-DOS that it offers.

The valuable information behind all this history is that Paterson, in creating 86-DOS, chose his operating system interface conventions to match those used in CP/M. Independent software developers had by that time written a large variety of applications to run under CP/M. It was important to Seattle Computer Products' marketing of its 8086 card that the predominant suppliers of existing software written for CP/M would find it relatively easy to take advantage of the 8086's superior capabilities. As it happens, when IBM entered the personal computer field, a host of new software was written for the IBM PC. None of this software had need of the CP/M conventions.

Perhaps the most rewarding aspect of Paterson's work, for himself as well as for us, was that he was able to provide an operating system freed of the limitations inherent in CP/M. He also got some of his wish list for operating-system capabilities taken care of. The result is a system that is several steps beyond CP/M—competent instead of irritating in its handling of error conditions, and friendly to the user.

SUMMARIES OF THE CHAPTERS

Chapter 1 is an introduction to operating systems in general and to MS-DOS in particular. The capabilities of MS-DOS are described from the point of view of the programmer and the applications user. Relevant aspects of the 8086/8088 architecture are introduced, followed by an overview of the utility routines that MS-DOS makes available to programs via function calls. The software components that make up MS-DOS are also discussed.

Chapter 2 discusses files in MS-DOS: their structures and capacities, the tools available for file creation and deletion, and data entry and manipulation. The tree-structured file handle-based file-access mechanisms are discussed, as well as the file-access strategy of version 1, which is based on file control blocks.

Chapter 3 gives a thorough description of the use of disk space in MS-DOS, including how the disk is organized into numbered sectors, what the capacities of some variously formatted disks are, a description of the disk access functions, and how the first few sectors of a disk are used for holding the MS-DOS software components. Chapter 3 also covers the use of file allocation tables for the allocation of disk space to files.

Chapter 4 discusses the issues involved in reading single characters and strings of characters from the keyboard. It also shows how to make use of the keyboard access functions provided by MS-DOS. The key-reassignment feature is also described.

Chapter 5 introduces two of the display types (monochrome and color/graphics) that accompany the IBM PC compatibles. It describes how to use both the MS-DOS and the ROM BIOS functions to address the screens. Chapter 5 includes the generic MS-DOS function calls and the screen-control sequences provided by the ANSI.SYS device drives.

Chapter 6 introduces the serial and parallel port-handling capabilities; they exist as function calls within MS-DOS and as part of the CONFIGURE or MODE [PCCs] command.

Chapter 7 is about dates and times in MS-DOS: their different representations, the function calls used to access them, and how MS-DOS maintains the time and date.

Chapter 8 is about two separate subjects, error handling and memory management. The first part of the chapter shows how in version 3 error condition interpretation is centralized and shows previous versions' approaches to error reporting. The second half of the chapter discusses memory management and memory-resident programs.

Chapter 9 discusses the networking features of MS-DOS. It introduces the concepts involved in programming for network operations, describes the new network-related function calls, and shows how some existing function calls have changed to accommodate network operations.

Chapter 10 begins the second part of the book with a discussion of MS-DOS at the user's level. Chapter 10 is a grab-bag of useful introductory information. Included are descriptions of MS-DOS keyboard features, a note on what parts of the DOS manual are most important to read, how MS-DOS handles disk errors, and an introduction to a few essential MS-DOS commands.

Chapter 11 is a quick look at EDLIN, the MS-DOS editor with characteristics that endear it to so few.

Chapter 12 talks about files from a user's point of view. It discusses how to set up a filing system and how to choose file names. It also talks about file capacities and the tree-structured filing system.

Chapter 13 shows you how to make use of the tools that MS-DOS provides to configure your system for best performance. It also shows you how to set up your hard disk, and how to look after the files you add to that disk.

Chapter 14 is all about batch operations, i.e., how to use the MS-DOS batch capability to save you much repetition and to have your computer perform sequences of commands without your intervention.

Chapter 15 gives the user's view of the displays that you might want to use with your MS-DOS computer. It discusses two of the types offered by the IBM PC compatibles, monochrome and color/graphics. It shows color selection on the color/graphics display, and how well programs written for one type of display adaptor will run on the other. As with Chapter 5, much of the information here is specific to the IBM PC and its compatibles.

Chapter 16 shows you how to use the serial port to redirect output from the parallel printer port to printers attached to the serial port, and how to connect with information retrieval services.

Chapter 17 shows you how to use DEBUG: what it can do for you, how to patch a program using DEBUG, how to create short programs using the debugger, and how to use the debugger to look around MS-DOS.

Appendix A discusses the differences between the various computer manufacturers' versions of MS-DOS and especially the difference between MS-DOS and PC-DOS. It includes notes on the much-claimed and widely differing interpretations of IBM PC compatibility.

Appendix B talks about the differences between the various releases of MS-DOS version 1, and the various releases of versions 2 and 3.

Appendix C lists the function calls offered by MS-DOS, grouped by function and in numerical order.

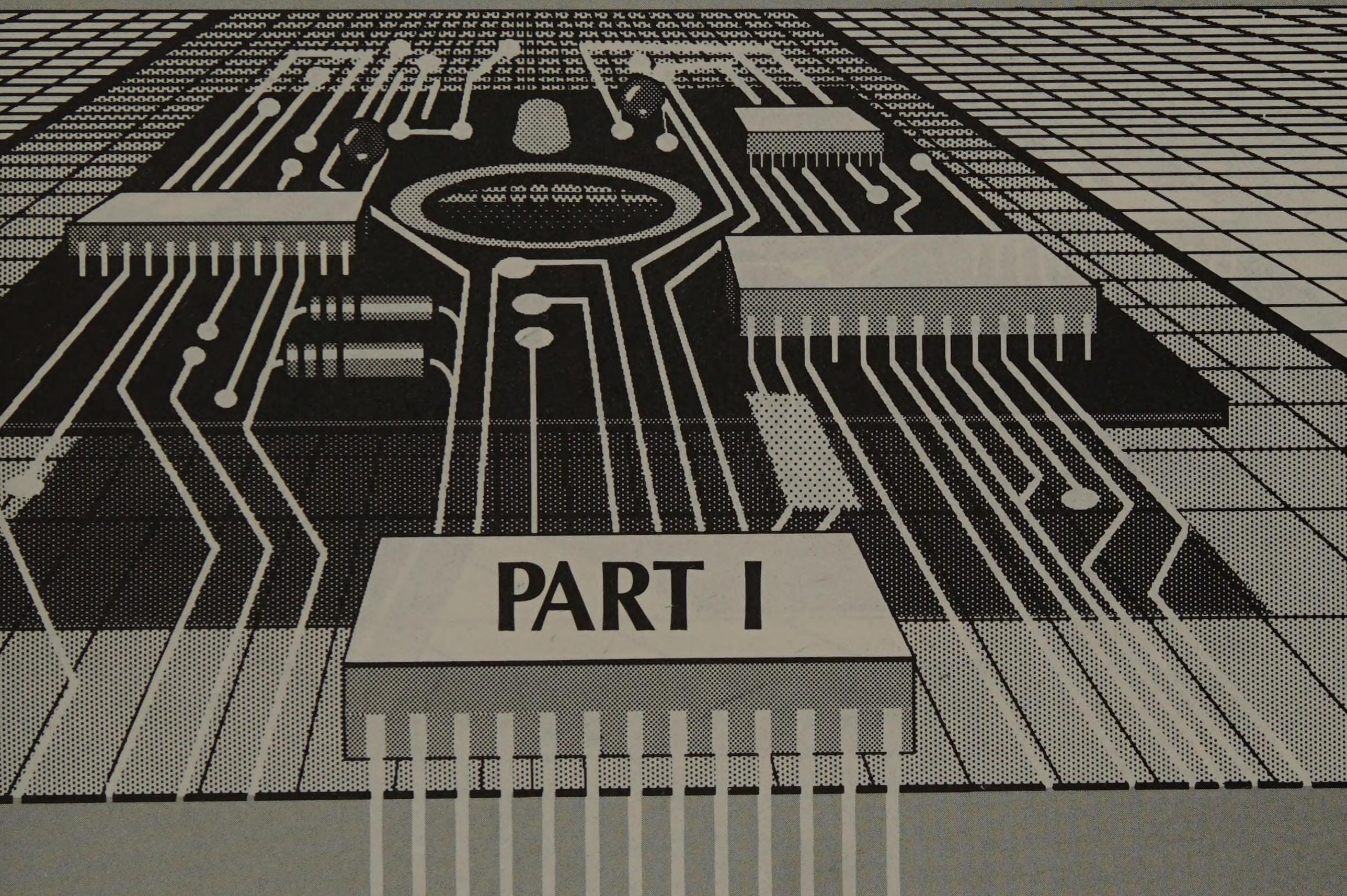
Appendix D shows how MS-DOS makes use of the first 32 interrupt vectors.

Appendix E includes memory maps of the 8086 address space as used by MS-DOS.

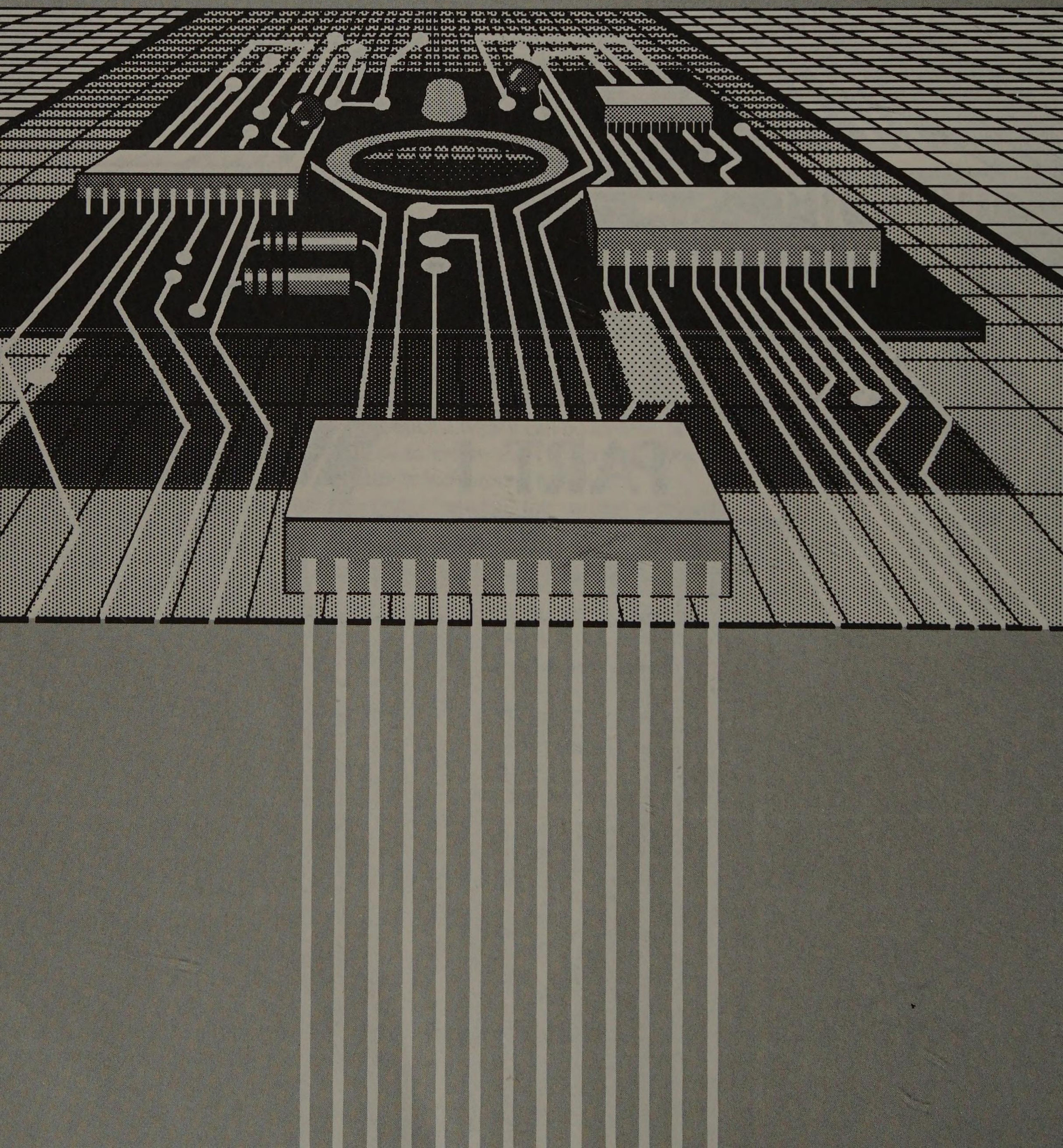
Appendix F provides tables that simplify the conversion from hexadecimal numbers to decimal numbers.

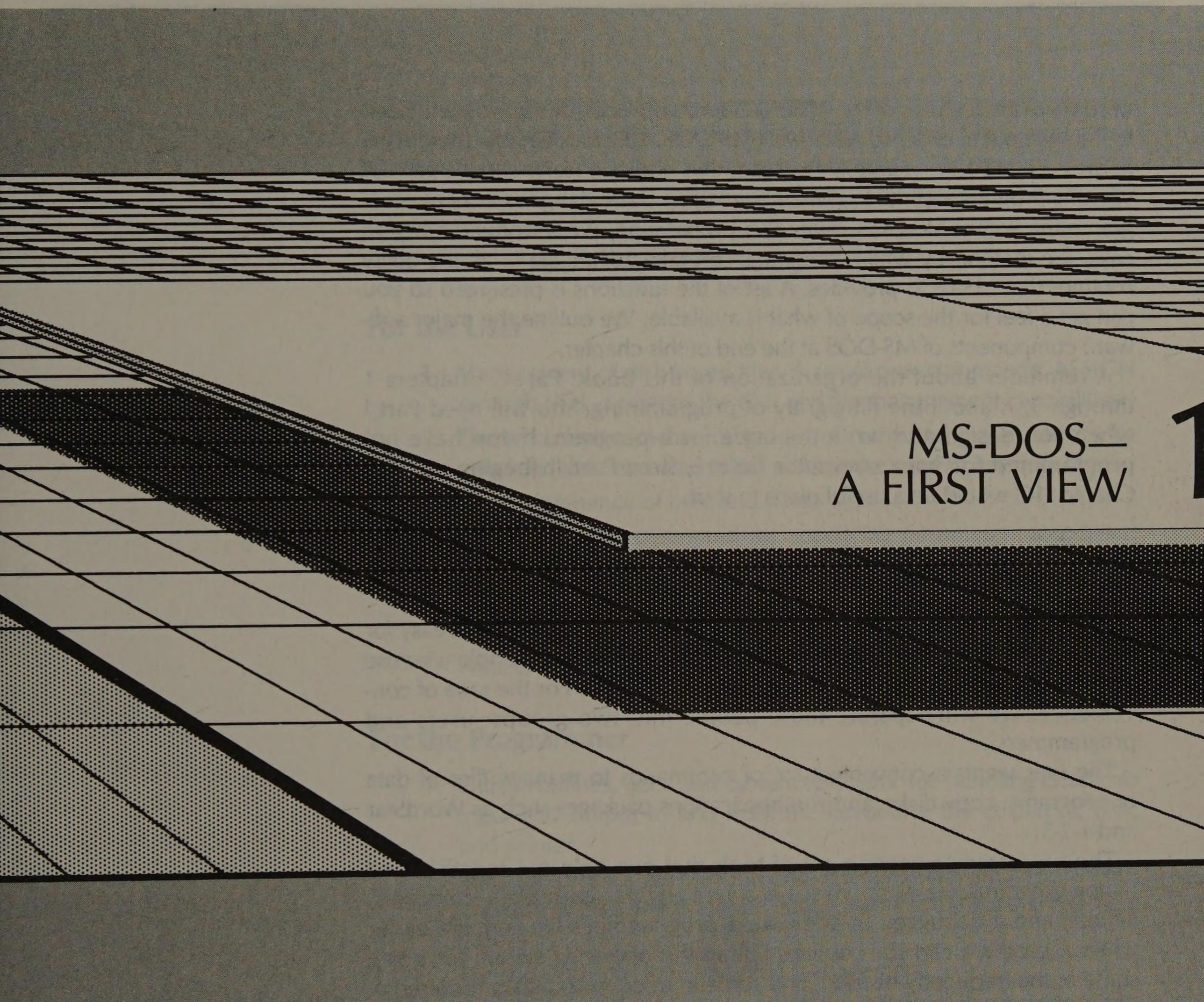
Appendix G is a summary of MS-DOS commands and their syntax.

THE PROGRAMMER'S HANDBOOK



PART I





MS-DOS— A FIRST VIEW

1

INTRODUCTION

By the end of this chapter, you will have a feel for the range of what MS-DOS can do for you as a programmer, and you will have enough background knowledge to make good use of the information presented in the chapters that follow.

Like other chapters, this one begins with a short summary of the topics covered. To give a perspective on MS-DOS, we start the chapter with a brief introduction to operating systems, then examine the fundamentals of MS-DOS. We introduce ROM BIOS, a small piece of an operating system that is built into the hardware of your computer. Next comes a description

of some of the features of the 8086/8088 microprocessor that you will need to know about in order to work with MS-DOS. You will see how memory is addressed and how registers are used, and you will learn about interrupt vectors, program segment prefixes, and a few other things essential to programming within MS-DOS. After this preparation, we introduce function calls, the method by which programs can take advantage of all the utility routines that MS-DOS provides. A list of the functions is presented so you can get a feel for the scope of what is available. We outline the major software components of MS-DOS at the end of this chapter.

A reminder about the organization of this book: Part I, Chapters 1 through 9, is about the nitty-gritty of programming. You will need Part I when you design and write the code for a program. If you have not programmed for your computer before, then Part II (beginning with Chapter 10) would be a useful place to start.

OPERATING SYSTEMS

An operating system is a set of software tools designed to make it easy for people and programs to make optimum use of a computer. People who use computers have different levels of needs and interests. For the sake of convenience, we will separate these people into two groups: users and programmers.

The *user* wants a convenient set of commands to manage files of data or programs, copy disks, and run applications packages such as WordStar and 1-2-3.

The *programmer* needs a set of tools that can help put together and debug programs. He or she needs program editors, assemblers, compilers, linkers, and a debugger. In addition, a programmer's life is made easier when a program can use routines within the operating system for doing some of the frequently needed (and sometimes complex) tasks required of any program, such as file manipulation and transfer of data to and from the various devices attached to the computer.

MS-DOS is an operating system designed for both users and programmers. You can do a lot with MS-DOS.

MS-DOS FUNDAMENTALS

MS-DOS is an operating system for 16-bit, 8086/8088-based computers. The name DOS is used instead of OS, because the programs that make up the operating system are designed with the assumption that files are to be found on the disk. The system needs a disk before any useful action can take place.

This operating system, MS-DOS, provides a variety of functions whereby a program can conveniently use the hardware it runs on. It also provides a set of tools for the user and programmer (often the same person) to keep track of programs and data, execute programs, and design and build new programs.

These are the fundamental capabilities of MS-DOS:

For the User

1. Management of disk-based files. A set of user commands, ASSIGN through VOL, provides the disk and file management capabilities.
2. The batch feature, whereby groups of command sequences can be saved and then executed at will without the need to respecify them.
3. The maintenance of time and date.
4. A configuration command for specifying parameters associated with the printer and the serial port, and display parameters [PCCs].
5. I/O redirection, pipes, filters, a form of print spooling, and the ability to print an image of the graphics display.
6. New to version 3, some network support features.

For the Programmer

1. Utility routines, accessed by function calls, for handling character input and output to and from the keyboard, the display(s), disk, and printer.
2. Function calls for file creation and deletion, data entry and access, and disk access. These are the components of the file management capability.
3. Function calls for time and date management.
4. A standard methodology for program initiation, program termination, and handling of the Break character (Ctrl-Break or Ctrl-C).
5. Program development tools: the assembler (in some versions), the linker, the debugger, and EDLIN, an extremely elementary editor.

The function calls are described a little later in the chapter. For now, let's look at some software that is not part of MS-DOS, but that MS-DOS uses a lot.

MS-DOS AND ROM BIOS

ROM BIOS is a piece of operating-system software built into the hardware of your computer. It is resident in read-only memory (ROM). It does

what the name, Basic Input/Output System, implies: it handles the basic input and output functions for the computer. Like MS-DOS, it contains routines to respond to function calls, and it does useful work for whoever calls those functions. MS-DOS uses many of the ROM BIOS functions, and ROM BIOS in some areas is more powerful than MS-DOS. If you are programming for an MS-DOS computer, it doesn't make any sense to look at MS-DOS without also looking at ROM BIOS.

In addition to I/O support, ROM BIOS usually contains

- the computer's primary bootstrap,
- routines that test and report errors in all the currently configured components of the computer,
- a set of patterns for generating the first 128 ASCII characters when the display is operating in graphics mode,
- time-of-day maintenance routines, and
- the print-screen utility.

ROM BIOS is useful only to programmers, not directly to users. It contains no features for accepting user commands, and it has no file-handling capabilities. ROM BIOS modules are also incorporated with some additional hardware that you might purchase for your PC, for example the IBM enhanced graphics adaptor. These ROM BIOS modules either replace or add to the main ROM BIOS.

The connection between MS-DOS and ROM BIOS is intimate but not indissoluble. MS-DOS uses a number of the features of ROM BIOS. MS-DOS could be rewritten to ignore the existence of BIOS and still appear to the user and programmer to be the same system. The question of interest to programmers is this: if ROM BIOS is in any way better than MS-DOS, why not use ROM BIOS instead? The versions of ROM BIOS found in the best IBM PC compatibles are functionally so close to the original IBM PC's ROM BIOS that direct use of ROM BIOS is appropriate. However, your program is more likely to be portable from **any** machine's MS-DOS to any other machine's MS-DOS if it does not use ROM BIOS.

Fully portable programs can be moved from computer to computer without difficulty. Portability from one machine to another is achieved when the language used to write the program is machine-independent, and when the operating system provides a buffer between the program and the unique characteristics of the hardware it runs on. Clearly, 8088 assembly language is not machine-independent, but MS-DOS has established a set of conventions for handling I/O. Programs that are to be portable between MS-DOS and PC-DOS, and between one manufacturer's version of MS-DOS and another's, have to use these DOS conventions.

ROM BIOS on occasion offers better facilities than MS-DOS for making use of unusual features of a computer. For example, the IBM PC compatibles often have two types of display adaptors; ROM BIOS has better functions for accessing them than does MS-DOS. Wherever ROM BIOS in the IBM PC compatibles is better, it is mentioned in the relevant discussions in this book. Let's take a closer look at some of the features of the Intel 8086/8088 family of microprocessors that MS-DOS is built around.

FEATURES OF THE 8086/8088 FAMILY

We cover here the essentials of what you need to know about the 8086/8088 family of microprocessors. The 80186, 80286, and 80386 are more powerful than the 8086/8088, but all 8086/8088 instructions run on the more advanced processors, and the basics described here apply to all variants of the Intel family. It is assumed that you have some experience or knowledge of programming at the assembly-language level, or will be learning more about it soon from another book. To be able to program within MS-DOS, you will need to know how memory is addressed, how registers are used, how to use interrupt vectors, and the purpose of a program segment prefix.

Memory Addressing and the Segment:Offset Format

The main microprocessor in the computer handles addresses in the segment:offset format. The 8086/8088 has a one-megabyte address space—1,048,576 bytes; it takes 20 bits to address one megabyte, so, clearly, more than one 16-bit word must be used to specify an address. The architects of the 8086/8088 resolved the problem of how best to use registers to address the memory by coming up with the segment:offset format. The segment register addresses a 64K chunk (segment) within the one-megabyte address space, and the offset register points within the 64K segment.

By convention, an address is expressed in hexadecimal digits, 4 bits to a digit; 20 bits can be expressed in five hex digits, so a 20-bit absolute address can be written as, say, B0000H (which is the address of the character in the top left corner of the monochrome screen in IBM PC compatibles). The absolute address B0000H is expressed in segment:offset format as B000:0.

You will quickly become accustomed to the segment:offset form of addressing; for those who are not familiar with it, here is more information about segments, followed by examples of converting segment:offset addresses to absolute addresses.

More on the Segment:Offset Format

A segment of memory contains up to 64K (65,536) bytes and always begins at a paragraph boundary. A paragraph is 16 bytes long and begins at

an absolute address that is a multiple of 16, so the address of a paragraph boundary always has the four low-order bits (the last hex digit) equal to zero.

You can view the segment part of a segment:offset address as a count of the number of paragraphs away from absolute address 00000H (the first location in memory). (Again, once you become familiar with segment:offset form, you will probably forget this interpretation of the segment portion of the address, and just consider it a starting point somewhere in the one-megabyte address space.) If you multiply the segment address by the length of a paragraph, that is, shift it left by one hex digit, you will find the absolute address of the segment.

The offset portion is a 16-bit address that refers to a byte anywhere within the area of memory whose starting address is defined by the segment register.

Converting Segment:Offset to Absolute Address

To convert an address in the segment:offset format to an absolute address, take the digits to the left of the colon, shift them left one digit, and add the offset. Let's look at some examples.

Conversion Examples The address, in the COMPAQ's BIOS communications area, of the word that holds the keyboard flag is 40:0017. Shift the segment portion, 40, left one digit to get 400. Add the offset, 0017, to get 417H, the absolute address of the keyboard flag. As another example, the address of the 8K ROM BIOS chip is F000:E000. Its absolute address is FE000H.

To illustrate a slightly more complex example, programs are loaded at offset 100H in the code segment. When you call DEBUG, the segment registers (see below) are set to the location of the program segment prefix (0CD0:0000 in COMPAQ version 3.1A). The program begins at 0CD0:0100, which is absolute address 0CE0H.

Segment Registers in the 8086/8088

Of the sixteen registers in the 8086/8088, four are set aside to be used as segment registers: the code-segment register, CS; the data-segment register, DS; the stack-segment register, SS; and the extra segment register, ES. The address of the current instruction is expressed by CS:IP, a combination of the code segment (CS) and the instruction pointer (IP). Similarly, the stack pointer is SS:SP, and the address of data is expressed by DS:DX. These are not the only valid combinations, as you will see when you look at the descriptions of the functions provided by MS-DOS.

Byte Registers in the 8086/8088

We have seen how some of the registers of the 8086/8088 are used. It is time to round out our knowledge of registers so that we can understand how MS-DOS uses them.

Each general-purpose register contains 16 bits, one word. Clearly, it is often convenient to deal in bytes rather than words. The 8086/8088 architecture allows for that by providing a way to specify that instructions operate only on either the low- or the high-order byte of a register. The four general-purpose registers (AX, BX, CX, and DX) can be addressed in this way. The fairly obvious ways of referring to each half of those registers are AH and AL, BH and BL, CH and CL, and DH and DL.

Let's look at another aspect of the 8086/8088 architecture that contributes to the organization of MS-DOS, interrupt vectors.

Interrupt Vectors

The 8086/8088 design supports software interrupts. Designers of earlier computers came up with the idea that some small part of memory should be set aside to contain the addresses of the various routines that handle hardware interrupts. Programmers quickly saw the potential in having a fixed address to use to get a particular function accomplished, and chatted avidly with the chip designers, so now we have locations set aside for software interrupts.

The first 1024 (400H) memory locations in the 8086/8088 are reserved for interrupt vectors. Each vector is the 4-byte address of the routine that services that interrupt. Of the 256 vectors, some are devoted to hardware interrupts; most are available for software interrupts. The majority of the first 16 vectors service hardware interrupts. ROM BIOS and MS-DOS between them use most of the next 32 vectors (up to about 30H).

Programs may change the contents of these vectors at will. Some are set at the time of system initialization. Each program can have its own way of handling a particular interrupt or function; if it does, it will redirect the appropriate vector to its own code. However, interrupt vectors that point to ROM BIOS tend to remain the same, because ROM BIOS provides many useful utilities.

The memory allocated to interrupt vectors allows for 256 separate vectors. MS-DOS provides well over 100 different functions that a program can call, so to restrict its use of the vector space, MS-DOS channels most of its function calls through interrupt 21H. The AH register is used to distinguish one request from another. Now let's examine a feature of program organization in MS-DOS, the program segment prefix.

Program Segment Prefixes

Each program that MS-DOS runs is given a program segment prefix (PSP). This is an area of memory, 256 (100H) bytes long, that serves as a communications area between the program and MS-DOS. The name *program segment prefix* is used somewhat in defiance of ordinary English syntax. The communications area begins at location zero of the segment allocated to the program, and it is a prefix to the program itself, which begins at offset 100H within the segment (the PSP extends from 0H to FFH).

The PSP is used to contain various items of useful information, including the size of the program segment (at offset 6), the addresses to which control passes under a number of different termination conditions, buffer space for the command line that brought the program to life (whose beginning coincides with the default disk buffer address), and a couple of tiny file control blocks, formatted to match the disk-drive designators and the file names of any arguments that were included in the command line. A diagram of the program segment prefix can be found in Chapter 8. The existence of the PSP explains why programs loaded into their segments begin at offset 100H instead of offset zero. MS-DOS creates the PSP when loading programs; the PSP does not exist on the disk.

Now let's look at function calls, through which a program can ask MS-DOS to perform a great variety of tasks.

FUNCTION CALLS IN MS-DOS

MS-DOS uses the interrupt vectors 20H through 27H. Interrupt 21H is called the function-request interrupt, and it is through this vector that most of your program's function calls will be made. The other vectors are for controlling program terminations and for reading from and writing to the disk without reference to MS-DOS files (the MS-DOS disk-access functions are described in Chapter 3).

In the area of function calls, all manufacturers' versions of MS-DOS are the same. If you write a program that runs successfully under MS-DOS, and it does not use ROM BIOS or directly access any unique features of a computer, it should run well under any other manufacturer's version of MS-DOS. (Unfortunately, however, programs written this way are sometimes inadequate when compared with programs customized to the hardware.) Let's see how programs can make use of these functions.

Calling Functions

To call a function, you must set the registers so that the function you are calling will know what to do, then simply execute the INT instruction.

This is a two-byte instruction of the form

INT *nn*

where *nn* (in hex) is the number of the interrupt vector. The machine language for INT is CD; the two hexadecimal bytes that form the interrupt 21H function request are CD 21.

Calling Functions from BASIC

You can also call MS-DOS functions from BASIC programs. Most versions of BASIC that work with MS-DOS are quite comprehensive in their capabilities, but now and again it is useful to be able to use one of these function calls. You can create small machine-language subroutines (using DEBUG) and use the CALL statement in BASIC to execute them without much difficulty. Passing arguments to the subroutine is a little more complex; it requires that you understand how to use the stack.

Let's look at the functions that MS-DOS provides within interrupt 21H.

Interrupt 21H Functions

Of the 100 or so different functions provided in version 3 under interrupt 21H, about 20 percent of them are used internally and exclusively by MS-DOS, and the other 80 percent can be called by your program to do something useful. (The actions of functions used internally by MS-DOS are not necessarily exclusively controlled by the value of the registers on entry, nor does Microsoft guarantee their functional consistency, so they should not be used by programs that are not part of the MS-DOS operating system.)

These functions are not grouped in any strict form; it seems that new functions are assigned numbers as the need arises. In the following table the functions are grouped alphabetically by what they do for us, rather than in numerical order (Appendix C has a list of these function calls in numerical order). These are the groups:

- Break-detection control
- Character device I/O
- Country-dependent information
- Date and time access
- Disk access (DTA access is found in file access)
- Error code access (new to version 3)
- File and directory access
- Memory management
- Network operations (new to version 3)
- Process handling (load and/or execute another program)

- Program termination
- Vector access
- Verify-read after write
- Version number access

Please take some time now to read through the following list of functions and their brief descriptions. When you are aware of the tools you have to work with, designing a program becomes much more rewarding, and it is more likely to lead to success. The function number, in hex, appears to the right of a brief description of the function. If you want to know more about one or a group of functions, use the index to find where it is described in more detail.

Break-Detection Control

More extensive checking for the Break character (Ctrl-C, or Ctrl-Break [PCCs]) is either on or off; this is controlled by function 33H (and by the BREAK command).

Character Device I/O

A variety of functions exist for sending and receiving characters to and from the devices on the system. Each function is device-specific although with I/O redirection the functions can apply to devices other than those for which they were originally intended.

Display (Standard Output Device)

Output, with Break-character detection	02H
Output, without Break-character detection	06H
String output	09H

File or Device I/O

Read CX characters into DS:DX buffer	3FH
Write CX characters from DS:DX buffer	40H

I/O Control Channel Access This is a general-purpose function for investigating the status of a device, and for reading and writing to the I/O control channel. The value of AL dictates the subfunction. AL values of 9 through 0BH are new to version 3.

Get device information (AL = 0)	44H
Set device information (AL = 1)	44H
Read CX bytes into DS:DX buffer (AL = 2)	44H

Write CX bytes from DS:DX buffer (AL = 3)	44H
Read CX bytes from BL drive number (AL = 4)	44H
Write CX bytes to BL drive number (AL = 5)	44H
Get input status (AL = 6)	44H
Get output status (AL = 7)	44H
Determine if device supports removable media (AL = 8)	44H
Determine if logical device is local or remote (AL = 9)	44H
Determine if handle is for local or remote device (AL = 0AH)	44H
Set lock conflict retry count (AL = 0BH)	44H

Keyboard (Standard Input Device)

Input, with Break-character detection	01H
Attempt input (if DL = FF)	06H
Input, without echo or Break-character detection	07H
Input, without echo but with Break-character detection	08H
Get string into DS:DX + 2	0AH
Check status (AL = FF if character ready)	0BH
Clear buffer, invoke 1, 6, 7, 8, or A above	0CH

Printer (Alias Parallel Port) I/O

Output character in DL	05H

Serial (Alias Asynchronous Port)

Wait for input character	03H
Output character in DL	04H

Country-Dependent Information

Get currency symbol, date format, etc.	38H
Get and set currency symbol, etc. (new to version 3)	38H

Date and Time Access

These functions get and set the system time and date. The function that accesses a file's time and date is shown in Chapter 2. The time and date formats are discussed in Chapter 7.

Get date into CX:DX	2AH
Get time of day into CX:DX	2CH
Set date from CX:DX	2BH
Set time of day from CX:DX	2DH

Disk Access

Most disk accesses will be generated by the file-access functions. The following functions are available for more unusual circumstances.

Disk reset, flush file buffers	0DH
Read CX sectors into DS:BX	Interrupt 25H
Select default disk drive	0EH
Write CX sectors from DS:BX	Interrupt 26H

Error Code Access

Get extended error code	59H

File and Directory Access Based on File Handles

These are the preferred file access functions. File identification is by ASCIIZ string and file handle. The functions access any address within the address space of a file and manipulate files within a tree structure of files. As you can see, the majority of function calls in MS-DOS have something to do with file access.

Whole-File Access

Create a file	3CH
Create a new file	5BH
Create a unique file	5AH
Open a file	3DH
Close a file	3EH
Delete a file	41H
Rename a file	56H

Reading and Writing Data

Read from a file (or device)	3FH
Write to a file (or device)	40H
Set address within a file	42H

File Handle Manipulation

Get another file handle for the same file	45H
Point existing file handle to another file	46H

Finding Files in the Directory

Search for first entry	4EH

Search for next entry	4FH

Directory Manipulation

Make a directory	39H
Remove a directory	3AH
Change to another directory	3BH
Get current directory	47H

Utilities

Parse the file name	29H
Set the disk transfer address (DTA)	1AH
Get the disk transfer address	2FH
Get the disk parameters (free space, etc.)	36H
Get or set a file's attribute	43H
Get and set the file's date and time	57H

File and Directory Access Based on File Control Blocks

The following set of functions originated with version 1 and, for compatibility purposes, is available in all later versions. These functions require the use of correctly prepared file control blocks that specify block numbers, record numbers, and record lengths. New programs should not use these functions.

Whole File Access

Create a file	16H
Open a file	0FH
Close a file	10H
Delete a file	13H
Rename a file	17H

Sequential Record Access

Read a record	14H
Write a record	15H

Random Record Access

Read single record	21H
Read multiple records	27H
Write single record	22H
Write multiple records	28H
Set random record field in the FCB	24H

Finding Files in the Directory

Search for first entry	11H
Search for next entry	12H

File Size

Read the file size	23H
Change the file size	28H, CX = 0

Memory Management

These functions allocate and deallocate memory in units of paragraphs (16 bytes).

Allocate BX paragraphs	48H
Change size of allocated memory block	4AH
Free allocated block (segment in ES)	49H

Network Operations

These version 3 functions are relevant when file sharing is in effect.

Lock and unlock file region	5CH
Get local computer ID (AL=00)	5EH
Specify printer setup string (AL=02)	5EH
Get printer setup string (AL=03)	5EH
Device redirection (AL=03)	5FH
Get redirection information (AL=02)	5FH
Change sharing retry count (AL=0BH)	44H

Process Handling

Execute (or load) a program/process	4BH
Terminate a process (EXIT)	4CH
Get completion code of terminated process	4DH

Program Termination

Use the first two functions rather than the last three.

Terminate, return completion code	4CH
Terminate, stay resident (KEEP)	31H
Terminate, no completion code	00H
Terminate, no completion code	INT 20H
Terminate, stay resident	INT 27H

Vector Access

Two central routines for accessing the interrupt vectors are offered:

Get interrupt vector value	35H
Set interrupt vector	25H

Verify-Read after Write

It is possible to tell MS-DOS to read and check for accuracy anything that it writes to disk, using these functions (or the VERIFY command):

Select or deselect read-after-write verification	2EH
Get verify status	54H

Version Number Access

Get the version number	30H

All these functions are described in Part I of this book, in the appropriate chapters. The descriptions show how your program can call the function, as well as the results (normal and abnormal) of calling the function.

The functions report their status on return in various ways. In functions that originated with version 1, the AL register is often used to pass error or completion codes. In version 2, an attempt was made to standardize error reporting; the carry bit is used to indicate whether an error occurred (set if error), and the AX register value corresponds to an entry in a table of errors. In version 3, extended error codes are provided via function 59H. Chapter 8 discusses error reporting in more detail.

Let's now look briefly at the programs that make up MS-DOS. The file structures, disk layout, and data I/O are all described in the chapters that follow.

THE COMPONENTS OF MS-DOS

MS-DOS consists of four separate components: the disk bootstrap, the basic input/output interface, the file and disk manager, and the command processor. The names of these software components vary from one manufacturer's version of MS-DOS to another. The description of each component shows some of the names used.

The Disk Bootstrap

The disk bootstrap is found on the first sector of every MS-DOS diskette, and on the first sector of the partition allocated to MS-DOS on a hard disk. It is put there by the FORMAT command.

When the computer is powered on, after ROM BIOS completes its equipment-check routines, and assuming all is well, ROM BIOS reads the first sector of the disk (the disk bootstrap) into memory (at location 7C00H [PCCs]) and executes it. The disk bootstrap program takes over from there. Its task is to load the basic input/output interface software into memory (600H [PCCs]). The disk bootstrap checks that it recognizes what it has read, and continues from there to read in the file and disk manager following the basic input/output software. If it does not recognize the program, it displays a message saying that the disk is not a system disk, and asks you to please load another, then hit a character to continue. The disk bootstrap has served its purpose once the I/O and file-manager software are in memory.

The Basic Input/Output Software

This is the interface between the heart of the operating system and the I/O capabilities of the computer. This program's generic name is IO.SYS, but manufacturers can change it; HPBIOS.COM and IBMBIO.COM are file names you may see. Where ROM BIOS exists, IO.SYS is largely an interface between the file and disk manager (the heart of MS-DOS), and the computer's ROM BIOS. If you use DEBUG to look at the first few locations of IO.SYS, you will see a series of JMP instructions that branch to routines for keyboard input, character output, RS-232 serial port I/O, output to the printer, and time-of-day handling.

The File and Disk Manager

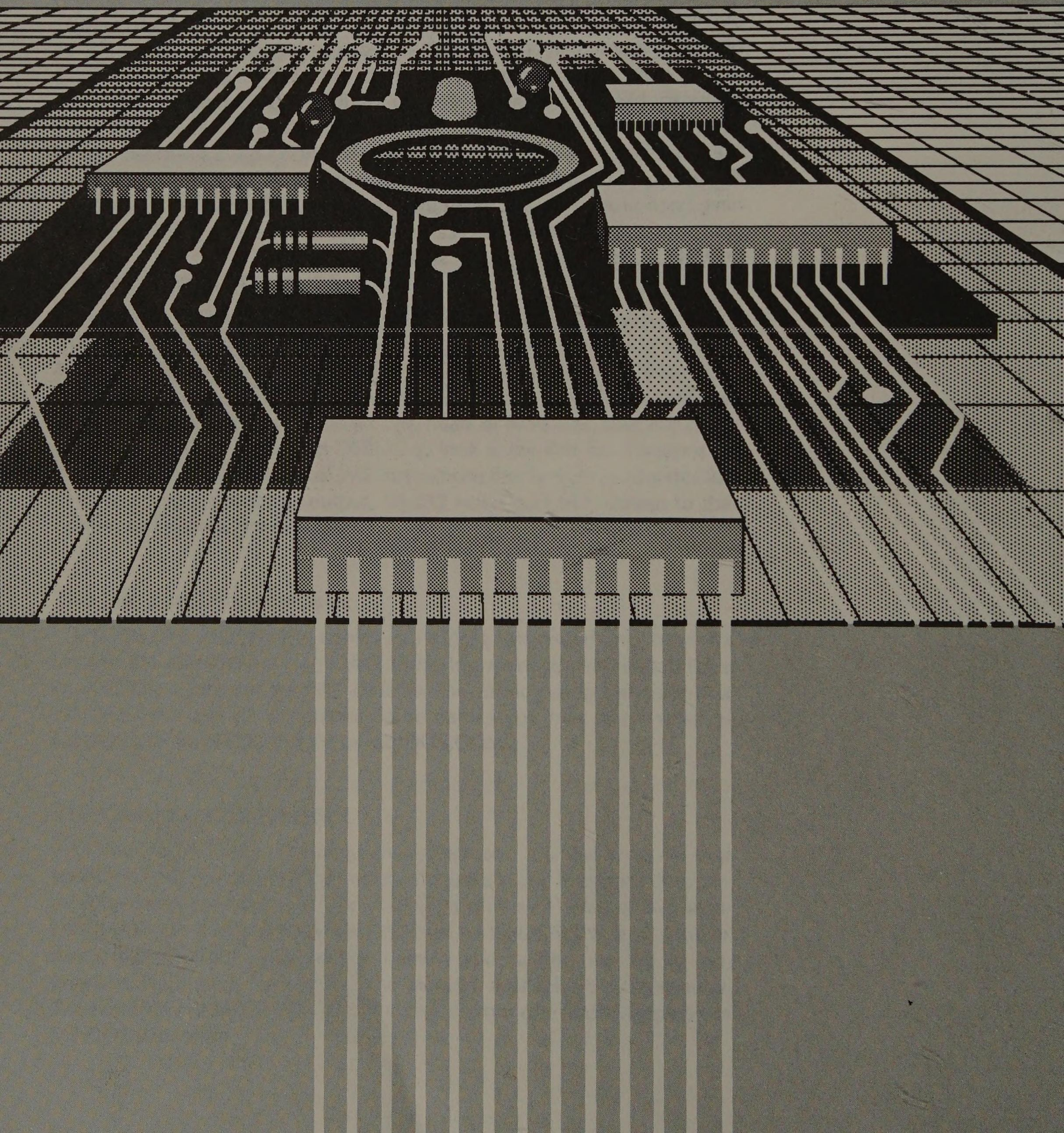
MSDOS.SYS (the generic name) is the program containing the code for the file management and disk buffering management capabilities of MS-DOS. It also contains the code for the various utility functions that MS-DOS provides for programs. Some names you may see instead of MSDOS.SYS are MSDOS.COM or IBMDOS.COM.

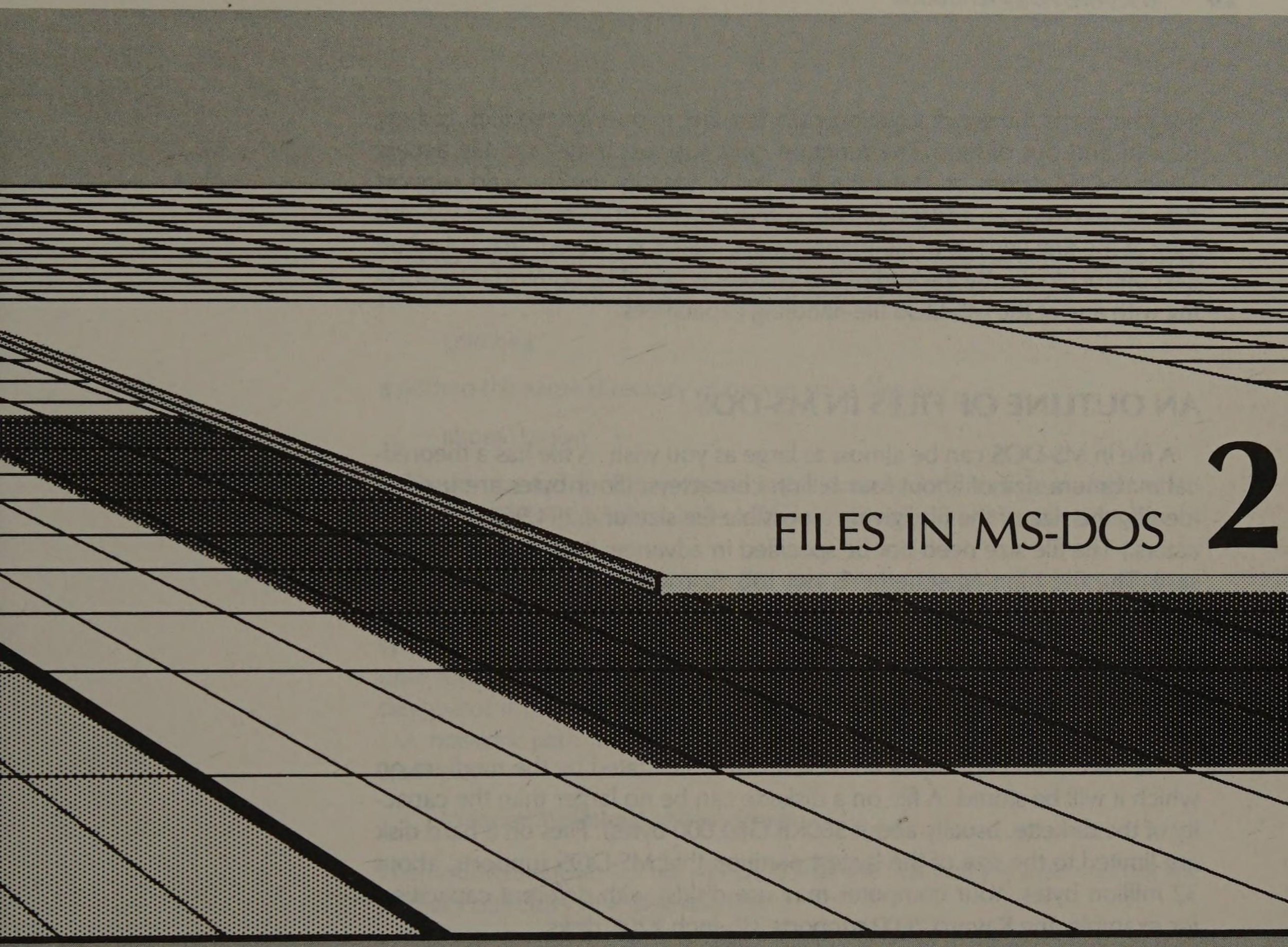
The Command Processor

COMMAND.COM is the command processor. It is the program that sends the system prompt to the display and looks at and interprets the characters typed at the keyboard. The command processor exists as two separate parts: one that is always resident, next to MSDOS.SYS, and another that is loaded into the high part of memory and may be overlaid by programs that occupy or use most of the memory available in the computer. The name COMMAND.COM is not changed from one implementation of MS-DOS to another.

SUMMARY

This chapter has provided introductory information about MS-DOS. An outline of the general purpose of MS-DOS has revealed that MS-DOS is an operating system that can be used with ease and success by programmers as well as by people who mostly need to run applications software. The necessary background information on the 8086/8088 microprocessor has been covered: how its address space is addressed, how its registers are used, and the software interrupt feature. The ROM BIOS has been introduced, as well as the four software components of MS-DOS. Brief notes have been presented, describing the purpose of all the functions that programs running under MS-DOS can call. Now it's time to explore the heart of MS-DOS, its file-management system.





FILES IN MS-DOS 2

INTRODUCTION

This chapter describes the two separate strategies for file handling and data access in MS-DOS. All current releases of MS-DOS base file handling on an ASCIIZ string that designates the file, and data access on an address within the file's address space. Version 1 uses file-handling and data-access techniques that are based on file control blocks and records within files.

We begin the chapter with a brief outline of files in MS-DOS. This outline includes notes on file capacities. We then describe how files are located using a *path* through the tree structure of files that MS-DOS supports, and discuss file attributes, access codes, and open modes. The major focus of

this chapter is the set of function calls that help you manage files and get data in and out of files. The function calls support individual file access using ASCIIZ strings or 16-bit file handles to identify the file, and support data access using an address of data within the file. To be complete, we also look at the first version of file management MS-DOS offered, that based on file control blocks. By the end of this chapter you will be comfortable working with any of the MS-DOS file-handling capabilities.

AN OUTLINE OF FILES IN MS-DOS

A file in MS-DOS can be almost as large as you wish. A file has a theoretical maximum size of about four billion characters. (Four bytes are used to identify the size of the file, giving a possible file size of 4,294,967,296 characters.) The file size need not be specified in advance; it expands as necessary. The file naming scheme is not too restrictive—an eight-character filename and a three-character filename extension. These is no mechanism for continuing files across disk boundaries.

The Capacities of MS-DOS Files

The maximum capacity of an MS-DOS file is dictated by the medium on which it will be stored. A file on a diskette can be no larger than the capacity of the diskette, usually about 360KB (360,000 bytes). Files on a hard disk are limited to the size of the largest partition that MS-DOS supports, about 32 million bytes. Your computer may use disk(s) with different capacities; for example, the Kaypro 2000 supports 3½ inch 720K disks.

LOCATING FILES IN MS-DOS

MS-DOS uses a tree-structured, alias hierarchical, filing system for its files. In a tree-structured system, each branch of the tree supports a number of files and can, in addition, support other branches of the tree. The file name identifies the file within the branch, and each branch is a directory of files within the branch. The same file name may exist in several directories of the tree. The route that takes you from the directory of the beginning of the tree (the root directory) is called the *path*. (If you are not familiar with tree-structured systems of file organization, see Chapter 12, "Your MS-DOS Files.")

The Path

The path is a series of directory names separated by backslashes (\). Each directory name takes you down another level in the tree, arriving

eventually at the file. The path begins at the root directory if the first character of the path is a backslash. For example,

```
\clothes\shoes\brown
```

may be the path to a directory of files for brown shoes on a disk devoted to personal possessions. Where the current directory is

```
\clothes
```

a path to the same directory of brown shoe files is

```
shoes\brown
```

The format for directory names is the same as for file names—eight characters, period, three characters, although the directory name extension is not often used.

The total length of a path from the root directory to the directory containing the file of interest cannot exceed 63 characters. Except where a network system is in use, the complete path leading to a file begins with a disk volume identifier, usually the disk-drive designator letter. Where a disk-drive designator is not used, the default drive is assumed.

A network path includes as the beginning of a tree a server name, for example,

```
\\server3\clothes\shoes\brown
```

The double backslashes tell MS-DOS that the first entry is a server, not the primary directory of a disk volume.

The ASCIIZ Path and File Name String

A full identification of a file in MS-DOS is an ASCIIZ string containing the path and the file name. It's an ASCIIZ string, rather than ASCII, because the string must be terminated with a zero byte. Functions that open files require this string.

File Handles

Using the path and file name ASCIIZ string to identify a file is clear, but is not always convenient. The *file handle,* a shorter identifier assigned by MS-DOS, is returned by the functions that open MS-DOS files. The file handle is a 16-bit identifier that is then used in place of the ASCIIZ path and file name string. Not by coincidence, device identifiers are also 16-bit values. I/O redirection, where files and devices are supposedly interchangeable, is more easily supported this way.

File Handles for the Standard Devices

The following table shows the handles assigned to five standard devices. (These devices are opened by MS-DOS. Your program does not need to open them before using them.)

Handle	Device
0000	Standard input device, input can be redirected
0001	Standard output device, output can be redirected
0002	Standard error device, output cannot be redirected
0003	Standard auxiliary device
0004	Standard printer device

Examination of the table shows why the first file opened after the system is booted is given a handle of 0005.

FILE ATTRIBUTES AND FILE ACCESS MODES

It is not always desirable to have all files open to access by all software and all users. Marking the file with an attribute is one way to permit the operating system better control over file access.

MS-DOS File Attributes

Each file in MS-DOS has an attribute, a one-byte flag that is maintained in the directory entry for a file. The purpose of the file attribute is to enable files to be marked for special handling by the operating system. For instance, IBMBIO.COM and IBMDOS.COM (sometimes called IO.SYS and MSDOS.SYS) are two files that exist on disks containing MS-DOS, but their names never show up in a directory listing. That is because their attribute marks them as hidden files. Each bit in the attribute flag indicates the presence or absence of a specific attribute. Figure 2.1 shows the correlation between bit and attribute.

The file attribute scheme becomes inadequate in multitasking situations, where different tasks can have different access rights. To provide adequate protection for files and data, MS-DOS provides *access modes*.

Access Modes

The access mode is only applicable when the file is open. You can use it to specify whether you want a file opened for reading, writing, or both. The ability to specify an access mode gives you additional control over the use of the file, beyond that specified by the attribute byte.

The access mode is specified at the time the file is opened using the open function (3DH). (A newly created file always has an access mode of read/write.) The three access modes are

Mode	Meaning
0	Open file for reading
1	Open file for writing
2	Open file for both reading and writing

Once the file has been opened with a restrictive access mode (read-only or write-only), the only way to access it differently is to close the file and reopen it with another access mode. Except for newly created files, the access mode will not override the effect of a read-only attribute byte. A request to open a file having a read-only attribute with access mode 1 or 2 will fail.

Open Modes

Now that MS-DOS supports networks, additional levels of access rights and restrictions become necessary. In version 3.1 the access code expands into the *open mode*. The original three access modes are maintained and sharing and inheritance modes are added to achieve the transformation from access mode to open mode. The sharing and inheritance modes are described in Chapter 9, "Networking with MS-DOS."

USING FILE-ACCESS FUNCTION CALLS

All MS-DOS file-access capabilities are found within interrupt 21H. Setting the AH register and the appropriate argument registers selects a particular function. Executing interrupt 21H activates the function. On return, if

Byte	Attribute Meaning
00	Normal read/write file
01	Read-only file
02	Hidden file
04	System file
08	Root directory entry containing volume label
10	Directory entry containing a subdirectory
20	The file's archive (dirty) bit

Figure 2.1: *Attribute Byte Values for Files and Directory Entries*

the function provides any status information, register AX contains it. (Note: The absolute disk read and write capabilities lie outside interrupt 21H, but they operate on files only by accident or by special design.)

We have seen how the file functions use one of two possible forms of file identification: either (1) an ASCIIZ string containing the path and file name (we will call this the path string); or (2) a sixteen-bit file handle.

The address of the path string (which must be terminated with a 00 byte) is passed in DS:DX. File handles are passed to functions in the BX register and received from functions in the AX register. File attributes and byte counts are passed in the CX register.

Error Reporting

Instead of each function call returning error conditions in a manner unique to itself, error-condition reporting and interpretation are somewhat unified.

The basic error indicator for the ASCIIZ/file handle functions is the carry flag. FCB-based file access functions return a nonzero value in AL, or do whatever they do to indicate error conditions.

In version 2, if the carry flag is set on return, AX contains the error code. The meaning of the errors returned in AX may be found by reference to a common table of error conditions. The first 15 entries in Figure 8.3 show the version 2 error return codes and their meanings.

The version 2 centralization of meaning has minor disadvantages. Several of the file-access functions will, on occasion, return error 5, meaning that access has been denied. It is nice to know that the operation failed, but a little more information about why it failed can be useful. Version 3 has supplied an answer. Function 59H—Get Extended Error—may be called to find additional error information.

WHOLE FILE ACCESS

The function calls within interrupt 21H provide for two separate levels of file access: (1) finding files and (2) working with a single file. These functions are listed in Figure 2.2. First we will examine the function calls for working with a single file—opening, reading, writing, and closing a file. After that we will examine the function calls for creating files and finding and altering directory entries for a file. The functions that operate on whole files are Create, Open, Close, Delete, and Rename.

Creating a File—Functions 3C and 5B

The Create function creates a new file or, if you use function 3C, truncates an existing file to zero length. If you care about the possibility of

Purpose	Function Number
Whole file access	
Create a file	3CH & 5BH
Create a unique file	5AH
Open a file	3DH
Close a file	3EH
Delete a file	41H
Rename a file	56H
Reading and writing data	
Read from a file (or device)	3FH
Write to a file (or device)	40H
Move the read/write address pointer	42H
Get file (or get/set device) status	44H
File handle manipulation	
Get another file handle for the same file	45H
Point existing file handle to another file	46H
Finding files in the directory	
Search for first entry	4EH
Search for next entry	4FH
Directory manipulation	
Make a directory	39H
Remove a directory	3AH
Change to another directory	3BH
Get current directory	47H
Utilities	
Set the read-after-write verify state	2EH
Get the read-after-write verify state	54H
Get the disk parameters (free space, etc.)	36H
Get or set a file's attribute	43H
Get and set the file's date and time	57H

Figure 2.2: *Summary of File-Access Functions*

destroying the contents of existing files, use function 5B, which fails if the file already exists but is otherwise identical to 3C. If you use function 3C, your program needs to confirm that a file does not exist before attempting to create it.

To create a new file put the ASCIIZ path string address in DS:DX, put the file attribute in CX, set AH equal to 5B, and execute interrupt 21H. This function opens the file if the creation was successful, and it returns the file handle for that file in AX. Further reference to the file must be made using the returned file handle.

After a successful file creation, the access mode is set to read/write, even if the attribute specified in CX was read only.

Creating a Unique File—Function 5AH

If your program needs a temporary file, it can use this function. MS-DOS is in a better position than your program to assign a unique identifier—it does that anyway whenever it returns a file handle in exchange for the ASCIIZ path and file name string.

To create a unique file, point DS:DX to a path (ending with a backslash "\") to the directory where the file will exist, set the attribute of the file into CX, set AH equal to 5AH, and execute interrupt 21H.

Opening a File—Function 3D

A file can be opened with one of three access modes: reading, writing, or both. The access mode is passed in AL. See "MS-DOS Functions under Network Operations" in Chapter 9 for details of the open mode that is passed in AL when MS-DOS is part of a network.

To open a file, set DS:DX to point to the ASCIIZ string containing drive, path, and file name; put the access mode of your choice in AL; set AH equal to 3D; and execute interrupt 21H. On return, if the carry is set, the error code is in AX. If the carry is not set, the file was successfully opened and the file handle for the file is in AX.

Closing a File—Function 3E

You need a file handle to close a file. To close a file, put the file handle in BX, set AH equal to 3E, and execute interrupt 21H.

The only error returned is code 6, which means that the file routines did not recognize the file handle. If the handle is recognized, the file buffers are flushed to disk and the file is closed.

Deleting a File—Function 41H

The Delete-file function uses the ASCIIZ drive, path, and file name string. To delete a closed file, point DS:DX to the ASCIIZ string, set AH to 41H, and execute interrupt 21H. An opened file cannot be deleted.

An error code 5 will occur if the file you are trying to delete has the read-only attribute. When this happens, the file's attribute byte must first be changed to 0 (using function 43H), and then the Delete-file function must be called again.

Renaming a File—Function 56H

Renaming a file requires two ASCIIZ drive, path, and file name strings. The drive, if used, must be the same in both strings.

To rename a file, point DS:DX to the file's current path and file name string, point ES:DI to the new path and name, set AH equal to 56H, and execute interrupt 21H.

With this function, a file can be detached from one directory and added to another directory. In effect, the file can be moved freely within the tree. This function will not move a file from one disk drive to another.

READING AND WRITING DATA USING FILE HANDLES

The file handle data transfer functions work for both files and devices. Three functions are provided: one for reading, one for writing, and one for setting the address at which the reading and writing will occur. A fourth function, status checking, can be used as an alternative way of detecting end-of-file conditions.

A file is an address space. Your programs specify an address within the file, then read or write data at that address. The organization of the data in the file is thus freed from any constraints (such as blocks and records) imposed by the operating system. (Note: This form of file-space addressing is not available in version 1 but can be crudely simulated there by using the random-access feature with a logical record size of one.)

Reading from a File—Function 3F

To read from a file a program must specify the number of bytes to be read and the address of the buffer into which the data will be read.

The address within the file from which to begin reading (the read/write pointer) need not be set unless random access is desired. The read/

write pointer can be set by the Set-address function (42H). When the file is opened, the address is set to zero. Each read or write action adjusts the pointer by the number of bytes read or written.

To read from a file, put the file handle into BX, put the buffer address into DS:DX, put the number of bytes to read into CX, set AH equal to 3F, and execute interrupt 21H.

On return, the number of bytes read is in AX. When the end of the file is reached, the carry is not set. The way to detect the end of the file (when reading) is to examine the AX register on return. If the AX value is not the number of bytes called for (AX is not equal to CX), end-of-file has been reached. The next read will get no more bytes.

Writing to a File—Function 40H

Writing to a file is a simple operation. To write to a file, put the number of bytes to write into CX, put the buffer address from which to write into DS:DX, put the file handle into BX, set AH equal to 40H, and execute interrupt 21H. The file will expand as necessary, unless the disk becomes full. The disk-full condition can be detected by comparing the value returned in AX, the number of bytes actually written, with the number of bytes the function was instructed to read. The read/write pointer is incremented by the number of bytes written.

Moving the Read/Write Address Pointer—Function 42H

Use this function to set the address at which random-access reading and writing is to occur. The function accepts both absolute and relative pointer-movement instructions, depending on the value of AL on entry. If AL = 0, the function interprets the value in CX:DX as an absolute address. If AL = 1, the address is set to the sum of the current address plus the value in CX:DX.

To set the read/write pointer, set AL to the absolute or relative adjustment control, set CX:DX to the address (absolute) or offset (relative), set the file handle into BX, set AH equal to 42H, and execute interrupt 21H. On return, the new value of the read/write pointer can be found in DX:AX. This function can be used to expand the file.

File Length Determination

The Set-address function has an ancillary mode of operation, which can be used to determine the length of a file. If AL = 2 on entry, the pointer is moved to the end of the file plus the offset. If the offset is zero, on return DX:AX will contain the address of the end of the file. If the offset is greater than zero, on return DX:AX will point beyond the end of the file. Since the

file is not expanded in this ancillary mode of operation, why the function would accept a nonzero offset is a mystery.

Get File Status—Function 44H

The I/O control function for devices can be used as an alternative method of monitoring whether the end of the file has been reached.

Function 44H is mostly used for device I/O control. It has eight subfunctions, three of which have meaning when the function is used on files. Subfunction 6 returns input status; subfunction 7 returns output status. For both subfunctions, a return value in AL of FF means that the end of the file has not been reached. AL contains a zero when the end of the file has been reached.

TRADING FILE HANDLES

Two functions provide ways for getting a second file handle for a file. The first function (45H) provides an entirely new handle, while the second function (46H) detaches a handle from one file and gives it to another.

Get a New File Handle for a File—Function 45H

To get an additional file handle for a file, put its file handle into BX, set AH equal to 45H, and execute interrupt 21H. On return, AX contains another file handle for the same file.

Programs that use these separate file handles for the same file must be written carefully. They must not try to use the two (or more) file handles as if they were for totally different files. All the file handles converge to the same table of a file's status information, which includes the read/write address pointer. For example, if two more-or-less independent parts of the same program are using different file handles for the same file, they should use the absolute mode of read/write pointer movement. If the two parts of the program independently make relative adjustments to the same address pointer, they will not be reading and writing data in the right places. Using the open mode to limit access can resolve some of these problems.

Point a File Handle to Another File—Function 46H

This is the other way of getting two file handles for the same file. The same cautions apply as for function 45H. To attach one file's handle to another file, put the file handle of the file you are willing to abandon into CX, put the other file handle into BX, set AH equal to 46H, and execute

interrupt 21H. On return, both file handles point to the same file. If the file to be abandoned was opened, this function closes it before attaching its handle to the other file.

FINDING FILES IN A DIRECTORY

The two functions covered in this section are search for first entry and search for next entry. The information that is returned when a matching file is found is returned in the current DTA (disk buffer), formatted as follows (the DTA address offsets are decimal values):

DTA Address	Content
0–20	Reserved for subsequent access calls
21	Attribute found
22–23	File's time
24–25	File's date
26–27	Low-order file size
28–29	High-order file size
30–42	Name of file terminated by zero byte

The last field (30–42) will be shorter than 13 bytes if the file name is less than twelve bytes.

Search for the First Entry—Function 4E

To find the first matching entry, point DS:DX to the ASCIIZ string containing the drive, path, and file name, put the relevant attribute into CX, set AH equal to 4EH, and execute interrupt 21H. On return, the first 43 bytes of the DTA will be set as noted in the preceding section.

Search for the Next Entry—Function 4F

To find the next matching entry, set AH equal to 4FH and execute interrupt 21H. The information in the current DTA is used to search for the next matching file.

MAKING, REMOVING, AND CHANGING DIRECTORIES

Like files, directories can be created and deleted. Also, a program sometimes needs the ability to move its basis of operations from one directory to

another. The following four functions provide the equivalent of four user-level commands: MKDIR, RMDIR, CHDIR (when used to change to another directory), and CHDIR with no arguments (when used to display the current directory).

Making a Directory—Function 39H

To make a directory, point DS:DX to the ASCIIZ string containing the drive and path that terminates with the name of the new directory, set AH equal to 39H, and execute interrupt 21H.

On return, if an error code 5 is indicated (access denied), the function was asked to create a copy of an existing directory and refused to do so.

Removing a Directory—Function 3A

To remove an existing directory, point DS:DX to the ASCIIZ string containing the drive and path, set AH equal to 3AH, and execute interrupt 21H.

MS-DOS will not allow you to remove a directory that contains any files or subdirectories, nor will it allow you to remove the root directory!

Changing the Current Directory—Function 3B

This is the equivalent of the CD path command, where the current directory becomes the one specified in the command. To change the current directory, set DS:DX to point to the ASCIIZ string containing the drive and path, set AH equal to 3BH, and execute interrupt 21H.

Getting the Current Directory—Function 47H

This function is the equivalent of the CD (CHDIR) command with no arguments. To get the current directory for a specific drive, put the drive number (0 = default, 1 = first drive, etc.) into DL, and point DS:SI to a 64-byte buffer where the function is to put the full path (but not the drive) name of the current directory.

UTILITY FUNCTIONS

MS-DOS provides a few utility functions to round out the set of tools required to manage its files. MS-DOS has the ability to verify its disk operations by reading information immediately after writing it. In addition, disk parameters (size, free space, etc.) can be read and a file's attributes and date and time may be set and read.

The Verify Capability—Functions 2E and 54H

Function 2E permits toggling of the read-after-write verify operation. For data that is vital to the state of health of the nation or planet, where you must be sure you have done whatever you can to ensure its correct recording on the disk, use this capability. Unless you use large block transfers, the time it takes to write to disk when read-after-write is enabled is on average close to triple the time it takes when it is not enabled. The disk-write failure rate is normally very low. However, this function could be useful if your computer is operating in a very noisy electrical environment.

To turn on read-after-write verification, three register settings are necessary. Set DL equal to 0, AL to 01, and AH to 2E, and execute interrupt 21H.

To turn off the verify operation, set both DL and AL equal to zero; set AH equal to 2E; and execute interrupt 21H. The complement to function 2EH is function 54H, which gets the verify state. Function 54H returns AL = 0 if verify is off and AL = 1 if verify is on.

Functions 2EH and 54H are the equivalent of the user-level VERIFY command.

Get the Disk Parameters—Function 36H

Function 1BH in version 1 returns the address of the file allocation table. Programs can then compute the free space on the disk by looking into the allocation table.

In later versions, the same function returns a pointer to a byte containing a copy of the first byte of the file allocation table. The file allocation tables for hard disks are not resident in memory, so it is not possible to hand out a pointer to them. Function 36H supplies immediately what had to be computed in version 1. To get the disk parameters, set DL to the drive number, set AH equal to 36H, and execute interrupt 21H.

On return, if the drive designator was valid (AX returned not equal to FFFF), DX contains the total number of clusters on the drive, BX contains the number of free clusters, AX contains the number of sectors per cluster, and CX contains the number of bytes per sector. This is all you ever need to know about a disk.

Get or Set a File's Attribute—Function 43H

To get a file's attribute, set AL equal to zero, point DS:DX to the drive, path, and file name ASCIIZ string, set AH equal to 43H, and execute interrupt 21H. The file's attribute is returned in CX.

To use this function to set a file's attribute, set AL equal to one and put the attribute in CX. The other arguments are the same.

Get and Set a File's Date and Time—Function 57H

In version 2, programs have the privilege of being able to set the date and time that is stored in a file's directory entry. Version 1 could read the date and time (just the date in version 1.0) from the FCB, but not set them. This function will either read or set the date and time, according to the value of AL on entry. If AL = 0, the date and time are read. If AL = 1, the date and time are set.

To access a file's date and time, pass the file handle in BX, set AL as needed, set AH equal to 57H, and execute interrupt 21H. The date is passed in DX, the time is passed in CX. The file's date and time are updated automatically by MS-DOS. Use of this function is not compulsory; MS-DOS updates the file's date and time whenever something is written to the file.

VERSION 1 FILE HANDLING AND DATA ACCESS

The system of files and file access implemented within MS-DOS version 1 is simple and barely adequate. MS-DOS version 1 neither supplies nor expects a specific system of organizing groups of files. It does not support a tree-structure filing system. It has the limitations that a single-sided diskette cannot contain more than 64 files and a double-sided diskette cannot contain more than 112 individual files.

A file and its current state are described by a file control block (an FCB) external to that file (see the section "File Control Blocks"). The access mechanisms were designed around the block and record concept of structuring a file, where the block is 128 records of whatever size you choose, up to a maximum of 64K bytes. Access may be either sequential or at random. Note that this method of file access is still available in later versions of MS-DOS.

Version 1 Programs under Later Versions of MS-DOS

A program that uses version 1 file-access calls will run fine in later versions, with a few possible exceptions. If a program uses the file allocation table (FAT) address supplied by function 1BH, it may have trouble when running in version 2 or later, because later versions return a pointer to a byte containing the FAT identification byte, not the address of the FAT itself. A program using many FCBs may not run in version 3 under some file sharing conditions. A program that leaves irrelevant values in arguments, for example, a program that does not clear unused bits when setting AL for function 3DH, may achieve unexpected results when running under, say, version 3.

Let's examine in more detail the version 1 approach to file structures.

THE STRUCTURE OF AN MS-DOS FILE—VERSION 1

A file is a series of data bytes, organized logically into records within blocks. A block is a group of 128 logically contiguous records. A record is a group of up to 64K (65,536) logically contiguous bytes.

The concept of having blocks of records in a file can be useful, but the choice of 128 (a fixed quantity) as the number of records within a block is arbitrary and restrictive—it deprives the concept of most of its usefulness. At least there is flexibility in the choice of record length, or else we would be stuck with a design feature that is almost two decades out of date. The block/record structure is important, because the file-access capabilities provided by the version 1 MS-DOS routines are in these units. Let's examine file control blocks, so we can work with this file structure.

FILE CONTROL BLOCKS

A file control block is the interface between the MS-DOS file access mechanisms and the file of raw data on the disk. The FCB contains information on the structure of the file, its name and length, and the current address of both sequential and random-access operations. An FCB does not contain information about the location of the file on disk—that information is contained in the disk directory and in the file allocation table.

Before you can create, read, or, in fact, make any access to a file, a file-control block frame must exist for that file. You can do nothing with files unless you have an FCB. For example, you need an FCB to step through the entries in a directory.

Two types of FCBs exist: standard and extended. The standard FCB is 37 bytes long. It is shown in Figure 2.3. The extended FCB is simply a standard FCB with a seven-byte prefix to indicate that the file described has special attributes (a nonzero attribute byte). Figure 2.4 shows an extended file control block, with the three different attributes that version 1 accepts.

Note the odd arrangement of the high- and low-order bytes in the current-block, logical-record size, and file-size fields of the FCB. The random-access record-number field is handled strangely as well. (Note: Because a word of two bytes is normally represented with the low-order byte—the byte with the lower address—on the right and the high-order byte on the left, it is awkward to represent the contents of a software structure in ascending byte order. Be aware of this when you review the FCB figures, where the addresses of the bytes ascend from left to right. For example, in the high-order-word part—bytes 18 and 19—of the file-size field, note that the high-order byte is at byte 18, not byte 19.)

File control blocks do not necessarily have all fields filled in at any one time. They are pieced together from your program and by the routines

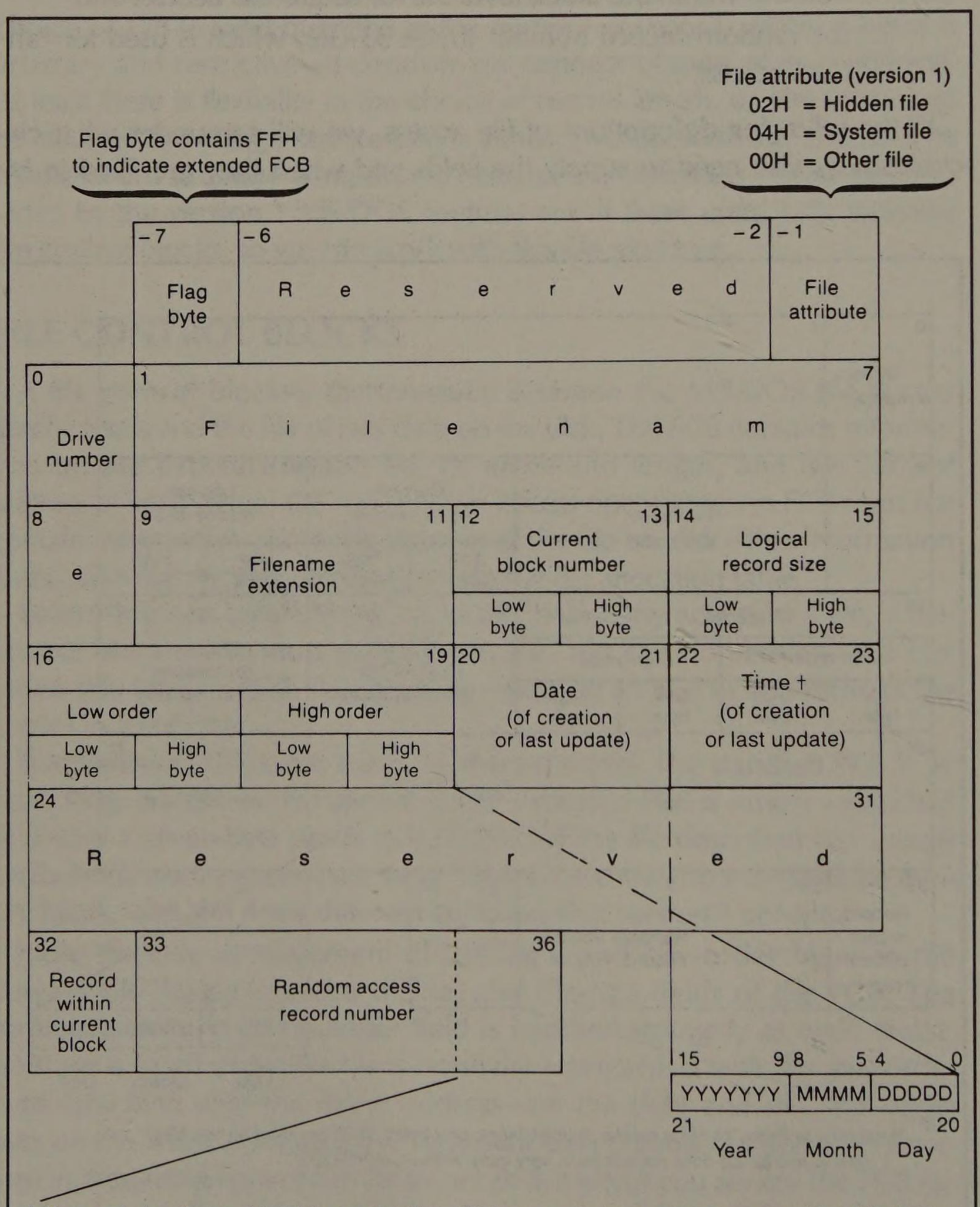
within the interrupt 21H function calls that take file control blocks as arguments. The fields that control access to the contents of the file are

1. the logical record size (bytes 14–15),
2. the current block number (bytes 12–13) and the current record number within the block (byte 32) for sequential access, and
3. the random record number (bytes 33–36), which is used for random access.

In the following descriptions of file access, we will see under what circumstances you need to supply the fields and when they are filled in by

Figure 2.3: *The Standard File Control Block*

MS-DOS. Before we look at file access (and now that you know what a file control block looks like), you need a place to create a file control block. But first we have to apologize for a little poor English, and then go ahead and use it.



* If record size (bytes 14-15) is greater than 64 bytes, only bytes 33-35 are used for this field.
† Time (bytes 22-23) does not appear in very early versions of MS-DOS

Figure 2.4: *The Extended File Control Block*

Unopened File Control Blocks

The text below often refers to "an unopened file control block." When an FCB contains only the disk drive, the filename, and the filename extension, it is the control block of an unopened file. The DOS manuals call this an "unopened FCB," and we shall adopt the same convention.

Choosing a Home for the File Control Block

To create an FCB, you need memory space. A standard location offered by MS-DOS is at address 5C in the program segment prefix (PSP). However, in keeping with the ideas that freedom and flexibility are desirable, that your program may need to manipulate more than one file at a time, and that a symmetric and systematic layout of FCBs is preferable to scattered FCBs, you could choose not to use the standard FCB address of 5C offered by MS-DOS.

Instead, you could set aside within your program space for a small set of FCBs, each with enough room to become an extended FCB should the need arise. An area of 4 × EXFCBlen bytes, where EXFCBlen is the length of an extended FCB, is sufficient for four FCBs. This allocation may avoid any fiddling around caused by the effects of the FCB outgrowing its previous space allocation, as has happened to the spaces allocated at 5C and 6C in the program segment prefix. The only times use of the FCB at 5C can be recommended are (1) during a search for a file using functions 11H or 12H, (2) when deleting a file using function 13H, and (3) perhaps when the program is first called.

When your program is called, MS-DOS will try to make filenames out of the first two arguments on the command line, putting them at 5C and 6C as embryo FCBs. It's nice for the operating system to be so helpful, but irritating because the file name at 6C is in the middle of the FCB starting at 5C, and when you use the 5C FCB it overflows into the default disk access buffer at 80H if you use record lengths of less than 64 bytes. Your program can create FCBs from ASCIIZ strings or command lines, using function 29H, which puts unopened FCBs wherever you want.

After that plug for a systematic approach to positioning file control blocks, let's move on to the function calls, which enable you to work with the MS-DOS file system.

ACCESSING A FILE AND ITS DATA IN VERSION 1

To make any file-access activity occur, all we have to do is make an adjustment to some field(s) in the FCB, set some registers, and then call the appropriate function. Let's look briefly at the subject of register settings.

Register Settings for Function Calls

In the descriptions that follow, we concentrate on the contents of the file control block and discuss how, why, and when the various fields within the FCB are adjusted. The register contents on the call to the file-access functions are discussed only briefly, because they vary little from one call to another.

The function to be called is specified in the AH register. The DS:DX registers point to the file control block. The CX register is used in the random-block read and write routines for a record count.

Now that we know which registers to use when calling file access functions, let's see which functions we have available.

Version 1 File-Access Functions

Figure 2.5 summarizes the file-access calls with their function numbers. Remember that these functions are also present in later versions and return the same results. To make a comparison with the function calls that make up the new file-access strategy provided by later versions, see Figure 2.2.

WORKING WITH A SINGLE FILE

Here we describe the set of tools you can use when working with a single file. A file can be opened and closed; its contents can be accessed sequentially or at random. If a file is accessed at random, either single or multiple records can be read or written. Let's see how to open a file that already exists.

Opening a File—Function 0F

A file must be opened before any data transfer can take place. An unopened FCB containing a drive specification, a filename, and a filename extension is all you need to open a file. To open the file, put the address of the FCB in DS:DX, put the value F in AH, and execute interrupt 21H. If you have reason to use an extended FCB, DS:DX must point to the first byte of the FCB prefix (the flag byte).

The Open function looks for the file name in the directory, returning FF in AL if it can't find it. If it finds the file name in the directory, it supplies values for four fields in the FCB. It also modifies the drive number field if it was set to 0 (the default drive number) prior to the Open operation.

The Open function zeroes the current block number (bytes 12–13), sets the logical record size to a default value of 80H (128 decimal), and reads the file size and the date from the directory entry for the file. The Open function does not set byte 32, the record number within the block, nor

does it set bytes 33–36, the record number relative to the beginning of the file—the field that is used for random access. It would seem natural to initialize both the current block and the current record within the block to zero to give sequential access a good start, but your program has to make either one or both settings; the Open function does not do it for you.

Purpose	Function Number
Whole file access	
Create a file	16H
Open a file	0FH
Close a file	10H
Delete a file	13H
Rename a file	17H
Sequential record access	
Read a record	14H
Write a record	15H
Random record access	
Read single record	21H
Read multiple records	27H
Write single record	22H
Write multiple records	28H
Finding files in the directory	
Search for first entry	11H
Search for next entry	12H
File size	
Read the file size	23H
Change the file size	28H, CX = 0
Utilities	
Get the current drive number	19H
Get the Disk Transfer Address (DTA)	2FH
Parse the file name	29H
Set random record field in the FCB	24H

Figure 2.5: *Summary of Version 1 File-Access Functions*

Reading from and Writing to a File

MS-DOS will read from or write to open files only. Once a file is open, you are only three steps away from transferring data: a record size must be specified, either the two current record fields or the one random record field must be set, and a disk transfer address (DTA) must be provided.

Those three steps cover a lot of ground, so we should proceed in an orderly way. First we will discuss the record size and record selection fields of the FCB. Then we will examine the function for setting the DTA. After that we will cover the functions that do the reading and writing, and examine the meaning of the status indicators on return from the function.

The Record-Size Field

The logical record-size field (bytes 14–15) of the FCB is set to a default value of 80H (128 decimal) by the Open and Create functions. If you are happy with that value, no change needs to be made. If not, alter the word at byte 14 in the FCB to the value you want. It could vary from 1, a single byte per record, to a very large number of bytes per record, say 64K. Practicality dictates the upper limit.

The Current Block and Current Record Fields under Sequential File Access

The current block and current record fields need to be set at the beginning of sequential file access. They are bytes 12–13 and byte 32 of the FCB. The current block is set to zero by the Create and Open functions. If you want sequential access to begin at some point other than block zero, set this field to the block of your choice.

The current record field, byte 32, is not initialized except by your program. Set it to zero to start sequential access at the beginning of the file; set it to the value of your choice (in combination with the current block number) to begin access at any location other than the first record.

Each successful sequential file access (via functions 14H and 15H) increments the current record field. When the current record reaches 128, the current block field of the FCB is incremented. As long as sequential access continues in this mode, no further setting of these two fields by your program is necessary.

The Current Block and Current Record Fields under Random File Access

There is a connection between the current and random-access fields of the FCB that you need to be aware of: the random-file-access functions translate the random record field into the current block and record fields.

When a random access is performed, the sequential access mechanism loses its place in the sequence.

The Random Record Field

The random record field, at bytes 33–36 in the FCB, directs the file routines to the *n*th record in the file, where *n* is specified by the random record field.

This field is *not* changed by the single-record random read and write activities (functions 21H and 22H in interrupt 21H). It *is* changed by the random block read and write functions (27H and 28H). These routines take a record count (in CX). At the end of the transfer, the random record field is updated to match the address of the next record. (Note: The concept of dividing the file into blocks of 128 records has no relevance to random access. The random record field specifies the *n*th record from the beginning of the file. The use of "block" in "random block write" is unfortunate, since all it means in this context is that more than one record can be read at a time.)

One other way of changing the random record field is to call function 24H, which sets the random record field based on the values of the current block and current record fields. One last reminder—any random access will set the current block and current record fields, i.e., the fields that drive the sequential access mechanisms.

Reading a Record under Sequential File Access—Function 14H

The file has been opened, and the logical-record-size field (bytes 14–15) has been set, either by the Open function to 80H or by your program to another value. The current block field has been zeroed by the Open function and perhaps reset by your program. The current record field has been set, and the DTA setting function has been called with your buffer address. Now, to read a record from the file, set DS:DX to point to the FCB, set AH equal to 14H, and execute interrupt 21H. On return, if AL = 0, the transfer was made successfully and the record number (current block, current record) was incremented. If there were any problems, AL would contain one of the following nonzero error codes:

01	End of file, no data in record
02	DTA's available segment space not large enough, i.e., buffer too small; transfer terminated early
03	End of file, partial record read, filled with zeros

Writing a Record under Sequential File Access—Function 15H

The differences between this and the Read function are the direction of transfer, the setting of AH to 15H before execution of interrupt 21H, and

the return codes in AL, which are as follows:

00 Successful write
01 Disk full
02 Record larger than DTA's available segment space, i.e., disk buffer too small

The read-after-write check facility, controlled by functions 2EH and 54H, is available whenever you need confirmation that the data recorded is just what you want.

Reading Records under Random File Access—Functions 21H and 27H

There are two major differences between random and sequential file access. The random record field of the FCB determines which record is accessed, and in random file access a choice must be made between reading single records or blocks of records.

The random read function (21H) reads a single record, while the random block read function (27H) reads the number of records specified by CX. As we noted before, during random access the records are numbered from zero, starting from the beginning of the file, and the current block number is irrelevant.

As with sequential access, before a successful random record file access can be made, the file must be open, the logical record size set, and the DTA specified. In addition, the random record number field in the FCB (bytes 33–36) must be set. For a random block read, CX must be set to the number of records to read; it must not be set to zero!

Random Read (Single Record)—Function 21H For a single-record random read, set DS:DX to point to the FCB, set AH to 21H, and execute interrupt 21H. On return, AL will provide one of the following codes as status information:

00 Transfer successful
01 End of file; no more data is available
02 DTA's segment space inadequate; transfer ended prematurely
03 End of file; a partial record is available, filled out with zeros

Random Block Read (Multiple Records)—Function 27H For a multiple record random read, set DS:DX to point to the FCB, set CX to the number of records to be read, set AH to 27H, and execute interrupt 21H. On

return, AL will provide status information:

00 Transfer successful
01 End of file after reading a whole number of records
02 As many records as possible were read before disk buffer space was exhausted
03 End of file, the last record read is a partial record

FCB Fields Changed by Random Reads The current block and current record fields that drive sequential access are always set by random access. This may seem odd, but there are semiplausible justifications for it. When the single-record-access function 21H is used, the current block and current record fields are set to the sequential equivalent of the random record address. For multiple record access via function 27H, those fields are set to the sequential equivalent of the record that follows the last of the group of records accessed.

Setting the Random Record Field in the FCB—Function 24H

This utility converts the file address found in the sequential access fields of the FCB to a random record file address. To use it, set DS:DX to point to an opened FCB, set AH equal to 24H, and execute interrupt 21H. The random record field will be set to the same file address as the current block and current record fields.

Writing Records under Random Record Access—Functions 22H and 28H

The preparation for the random record write is the same as that for the random record read, except for the difference in the value of AH.

Random Write (Single Record)—Function 22H For a single-record random write, set DS:DX to point to the FCB, set AH to 22H, and execute interrupt 21H. On return, AL provides status information:

00 Successful transfer
01 Disk full
02 DTA of inadequate length; transfer ended

Random Block Write (Multiple Records)—Function 28H For a multiple-record random write, set DS:DX to point to the FCB, set CX to the number of records to be written, set AH to 28H, and execute interrupt 21H. On

return, AL provides status information:

00 Successful transfer
01 Insufficient disk space, no records written

Note: Do not call this function with a CX of zero! This function has a special purpose when CX = 0 on entry—it changes the file size! This capability is no doubt convenient in other circumstances, but it could lead to dramatically unfortunate results if the routine that computed CX (the number of records to write) came up with zero. In fact, a civilized and flexible (but sadly inexperienced) program may want to call the Write function with a CX of zero, expecting no records to be written. Using CX = 0 to change a file length is a bit like defining NUL: to be an erase-all function. Using function 28H to change the file size is described in the section "Changing the Length of a File" later in this chapter.

Closing a File—Version 1

Files need to be closed before the program terminates, to ensure that data that belong to the file but are still in memory buffers are flushed to the disk (or other devices) and that the directory entry is correctly updated. This applies to files that are being changed by either the sequential or random write functions.

Before updating the directory entry, the Close function checks that the file name in the FCB is still in the same place in the directory, and only proceeds if that is the case.

The file-closing function call is 10H. To close a file, you put the address of an opened FCB in DS:DX, set AH to 10H, and execute interrupt 21H. On return, AL provides status information:

00 File successfully closed
FF File name not found in correct position on the directory; directory not updated

If AL returns an FF, your program could execute its own version of the error retry facility (the Abort, Retry, Ignore? message) provided by MS-DOS, to give the user a chance to change diskettes.

FINDING, CREATING, AND DELETING FILES—VERSION 1

This section describes the function calls that find and operate on the directory entries for files. First we will discuss the functions that look for

specific file names in the directory, then we will discuss the Create, Delete, and Rename functions. A note on the utility that tells you the file size is tucked in at the end of this section.

As with any file-access operation within version 1 of MS-DOS, a file control block is an essential prerequisite to doing anything useful. As usual, the address of the FCB is passed to the function in DS:DX.

Finding Files in the Directory

The file name as written by the user may contain the question mark (?) or asterisk (*) characters. Function 29H, which parses command lines looking for file names, will translate the asterisk character into as many question mark characters as necessary. This slight digression into user-specified file names has been taken to show that when looking for a file name in the directory, you must allow for the possibility of finding more than one match with it. Because of this, two functions are allocated to finding file names in the directory: Search for the First Entry and Search for the Next Entry.

Searching for the First Entry—Function 11H

This function does just what the name implies, and more. It uses an unopened FCB, which consists of the disk drive number, a filename, and a filename extension. Either of the filename fields may contain question marks, and in version 3, asterisks are accepted.

This function steps through the disk directory and returns when it finds the first matching file name. In addition, a valid unopened FCB is created at the disk transfer address. This feature is convenient, because the original unopened FCB you supplied cannot be used to access the file unless you are willing to abandon the search for further matches. The newly created FCB at the disk transfer address is, in fact, a copy of the disk directory entry, suitably offset from the DTA to make it into an acceptable FCB. This is the FCB to use to access the file found by the search. The next function-call description, Search for the Next Entry, sheds more light on this.

To search a disk directory for the first matching file name, set the DTA to an address where an FCB can be usefully created, point DS:DX to an unopened FCB, set AH to 10H, and execute interrupt 21H.

On return, if AL = 0, a valid unopened FCB will exist at the disk transfer address. It will be a standard or extended FCB, to match the one supplied via DS:DX. The disk directory entry has been placed at FCB address one; the disk drive number has been placed at FCB address zero; and the prefix has been supplied (if necessary) to match the prefix that DS:DX pointed to before the call. This is the FCB to use to access the file. If AL = FF on return, however, no matching entry was found.

Searching for the Next Entry—Function 12H

This function is used to find the next matching directory entry only after the search for the first entry function has been successful. The calling parameters and return values (AL = 0 for success, and AL = FF for none found) are the same as those in function 11H.

The reason the FCB that is pointed to by DS:DX cannot be used for file access is that it is used by these stepping functions to keep track of their place in the directory. Disk operations would cause the searching functions to lose their place.

Renaming a File—Function 17H

The function to rename a file needs the usual unopened FCB to find the file(s) whose name is to be changed. On finding a candidate, the file name beginning at DS:DX + 11H replaces the file name in the directory. (Note: I've not yet managed to use this function to change a set of files called fred1, fred2, fred3 into alfred1, alfred2, alfred3 in one function call. If you find a way, please let me know.) To change a file's name, set DS:DX to point to a modified FCB, where the new file name exists at DS:DX + 11H, set AH to 17H, and execute interrupt 21H. On return, AL = 0 indicates success. AL = FF indicates either that no match was found or that the rename operation would have created a file name identical to one already existing in the directory, and it refused to do that.

Creating a File—Function 16H

To create a file, you must start with an unopened FCB. MS-DOS will do the work of putting together an unopened FCB for you. COMMAND.COM creates unopened FCBs at 5C and 6C when the program is called, and you may be able to use one of these. If you have available a command line with the file name in it, you can use function 29H, described later, to create an unopened FCB. COMMAND.COM puts a copy of the command line at 81H when the program is called.

Once you have the FCB, put its address in DS:DX, set AH to 16H, and execute interrupt 21H. The Create function will then look for a free (or matching) directory entry, initialize it to a zero-length file, open that file, and return with AL = 0. You get no warning that you may be destroying an existing file; it is up to you to have used other functions, such as attempting to open the file, if you care about that not happening. If the directory is full and the file does not already exist, no file will be created and AL will equal FF on return.

If you want the file to be marked with one or more special attributes (say, hidden and system), an extended FCB (rather than a standard FCB) must be

used. Here DS:DX contains the address of the first byte of the prefix (FCB address minus seven), which is the first byte of the extended FCB.

Note that the Create function provides an opened file.

Deleting a File—Function 13H

Again, an unopened FCB (extended or standard) is needed to delete a file. This function deletes from the directory all files that match the filename and filename extension in the FCB.

The file name may contain the wild-card question mark (?) in place of any of the characters in the filename or filename extension. If the file name does contain wild-card characters, all files that match will be deleted. This function does not operate like the search functions, where only the first matching file is processed.

To delete a file, set DS:DX to point to the unopened FCB, set AH to 13H, and execute interrupt 21H. On return, if a matching directory entry or entries were deleted, AL = 0. If no match was found, AL = FF.

Reading the File Size—Function 23H

A file's disk directory entry contains the file's size in bytes. While a file is open, its size is maintained in the file-size field of the FCB, also measured in number of bytes. However, the version 1 file-access strategy deals in records, not bytes, so function 23H returns the file size in terms of the number of records in the file, based on the logical record size in the FCB. An unopened FCB must contain a file name that uniquely identifies the file you are interested in, because the file size returned belongs to the file with the first matching entry in the directory. The file size is returned in the random record field of the FCB, bytes 33–36. To read the file size in terms of number of records in the file, set DS:DX to point to the FCB, set AH to 23H, and execute interrupt 21H. On return, AL = 0 if a match was found and the file size supplied; AL = FF if no matching entry was found.

Changing the Length of a File—Function 28H, CX = 0

This utility is provided as a subset of the random block write function (28H). Its action is to change the length of a file to the product of the random-record and logical-record-size fields of an opened FCB. If you want to set the file size to a number of bytes, set the logical record size in the FCB to 1.

To use this function to change the length of a file, set CX to zero. Then set DS:DX to point to the FCB, set AH to 28H, and execute interrupt 21H.

If the length is reduced, the disk space allocation routine releases clusters

that become free. If the length is increased, additional clusters are allocated as necessary. However, if the disk becomes full, AL = 01 on return, and the increase in length is not made.

For more on the allocation and deallocation of disk space to a file, see "Allocation of Disk Space to an MS-DOS File" in the next chapter.

FCB-RELATED UTILITY FUNCTIONS

A variety of utility functions related to file access are supplied. These functions round out the set that is necessary for access based on file control blocks.

Setting the Disk Transfer Address (DTA)—Function 1A

A disk transfer address is a disk buffer address, a memory location to read into or write from. The read and write routines won't read or write data beyond the end of the segment specified by the DTA, nor wrap to the beginning of it, so make sure the DTA has enough space between the offset portion of its address and the end of its segment to accommodate your largest record.

The disk transfer address is set by a call to function 1A (INT 21H) with DS:DX being the DTA. Later versions have function 2FH, which gets the DTA. It's always a good idea to be able to read anything you have written.

Get the Disk Transfer Address—Function 2F

Function 1A sets the disk transfer address; version 2 supplies a function that will read it. To get the DTA, set AH equal to 2FH and execute interrupt 21H. On return, ES:BX contains the current disk transfer address.

The file name parsing function, which creates an FCB from a file name, is most useful.

Parsing the File Name—Function 29H

This utility extracts the file name and disk-drive designator from a command line to make a ready-to-use, unopened, standard FCB.

The action of the Parse function is controlled by the bit setting in AL. Bits 4 through 7 of AL are ignored. Bits 0 through 3 operate as follows:

Bit	Value	Meaning
0	0	Leading separators terminate the parsing
	1	Leading separators are ignored

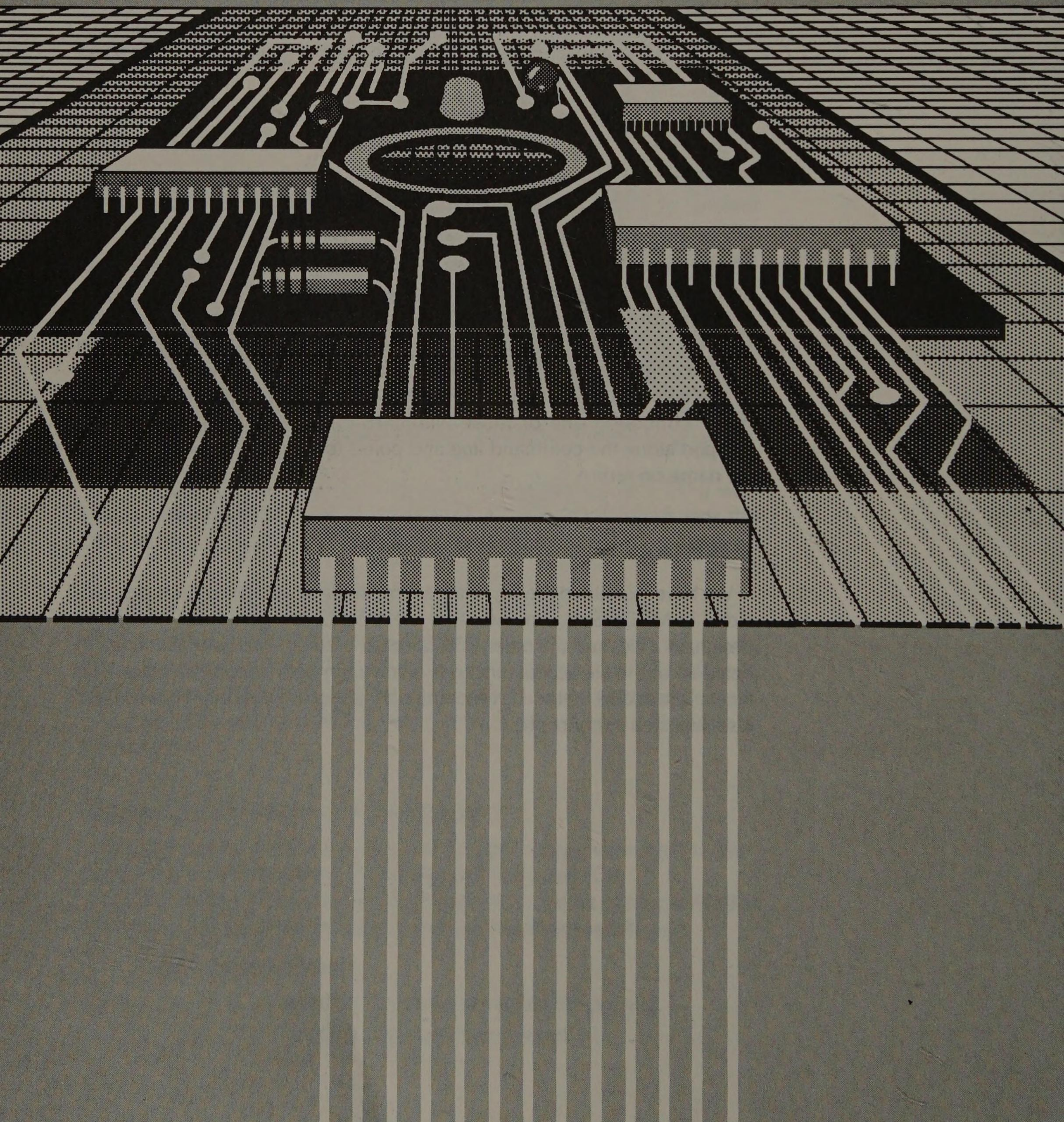
Bit	Value	Meaning
1	0	If no drive specified, drive number set to zero
	1	If no drive specified, drive number unchanged
2	0	If no filename specified, FCB filename set to eight blanks
	1	If no filename specified, FCB filename unchanged
3	0	If no extension specified, FCB extension set to three blanks
	1	If no extension specified, FCB extension unchanged

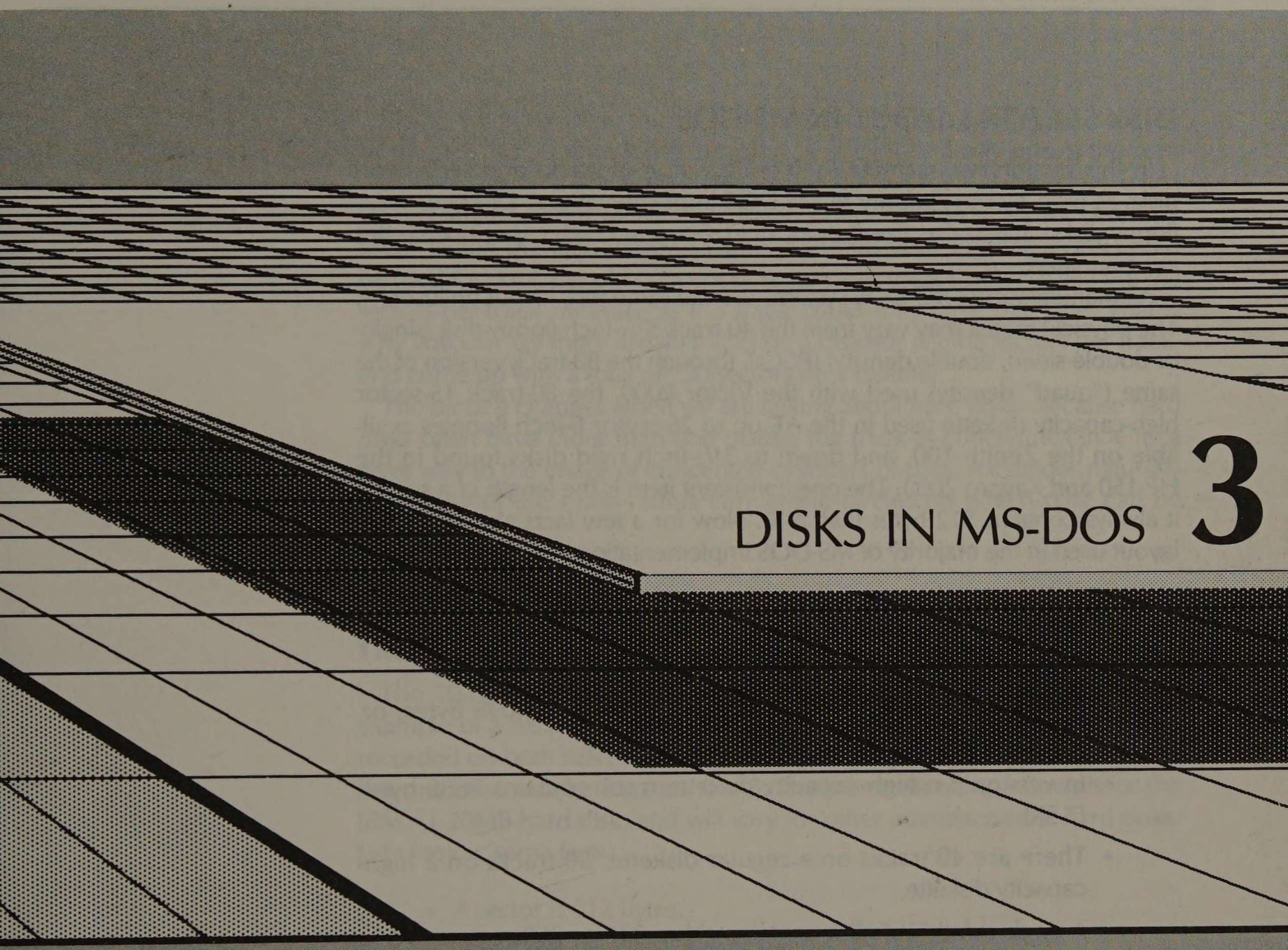
The normal setting for AL on entry is 0F. DS:SI points to the command line to parse, ES:DI points to the place to create the FCB. To call the function, set AH to 29H and execute interrupt 21H.

On return, AL does not provide the whole story. If it equals FF, the drive specifier in the command line was invalid. If AL = 00 or 01, the operation was successful, but the FCB has to be checked to see whether the first byte of the filename (ES:DI + 1) is not a space character. If AL = 01, the file name contained one or more wild-card characters (* or ?). DS:DI is stepped along the command line and points to the first character after the file name on return.

SUMMARY

We have examined the two different methods of file access strategies provided in MS-DOS: the method that uses ASCIIZ strings and file handles to designate a file, and the older FCB-oriented method. We have examined in detail all of the file-access functions involved in both access strategies. The file handle access method is recommended, because it is simpler to use and less restricted in concept.





3 DISKS IN MS-DOS

INTRODUCTION

This chapter describes how disks are used by MS-DOS. The chapter explains how disks are divided into sectors, tracks, and sides, and explains how each one is numbered. In addition, the chapter provides a list of the essential MS-DOS files found on an MS-DOS disk and examines the capacity of the space remaining for user files and programs. The allocation of disk space to an MS-DOS file using file allocation tables, and the file-independent disk-access functions available in MS-DOS are described. The chapter concludes with notes on programming for disk drives that are not compatible with those supplied by the computer manufacturer.

DISK SECTOR LAYOUT IN MS-DOS

In this section we examine the configuration of tracks and sectors on disks as used by the majority of MS-DOS implementations. Diskettes (alias floppy disks), hard (alias fixed) disks, rigid disks, and high capacity diskettes are examined.

All manufacturer's MS-DOS systems use the same disk layout philosophy. The physical media may vary from the 40-track 5¼-inch floppy disk (single- or double-sided, double density) [PCCs], through the 80-track version of the same ("quad" density) used with the Victor 9000, the 80-track 15-sector high-capacity diskette used in the AT, up to 26-sector 8-inch floppies available on the Zenith 100, and down to 3½-inch rigid disks found in the HP-150 and Kaypro 2000. The one consistent item is the length of a sector; it always contains 512 bytes (half a K). Now for a few facts about the disk layout used in the majority of MS-DOS implementations:

- There are 8, 9, or 15 sectors in a diskette track (version 1 recognizes only 8 sectors per track). High capacity diskettes use 15-sector tracks.
- A normal density diskette track contains either 4096 bytes, or, more often, 4608 bytes (4.5K).
- In version 3 a high capacity diskette track contains 7680 bytes (7.5K).
- There are 40 tracks on a regular diskette, 80 tracks on a high-capacity diskette.
- A *single-sided* 5¼" disk contains either 320 or 360 sectors of 512 bytes each, for a total of 163,840 or 184,320 bytes. A single-sided 5¼" disk is known as a 160K or 180K disk.
- A single-sided 3½" disk contains 1440 sectors of 512 bytes each, for a total of 737,280 bytes.
- A *double-sided* disk contains either 640 or 720 sectors of 512 bytes each, for a total of 327,680 or 368,640 bytes. A double-sided disk is known as a 320K or 360K disk.
- A *high-capacity* disk is always double-sided and contains 2400 sectors of 512 bytes each, for a total of 1,228,800 bytes.

Your computer's disk drives may have a different disk format, allowing it to hold more (or, unusually, less) information. The 40-track 5¼-inch diskettes have 48 tracks per inch; 80-track diskettes are used in drives that can address 96 tracks per inch. However, MS-DOS always uses a sector of 512 bytes, and all disks are addressed and allocated in units of sectors and clusters.

Before we move on to the hard-disk information, we need to see how the word *track* has a rather loose meaning. A track is basically one of the concentric circles on a disk on which data can be recorded. When people first started talking about sectors per track, only one side of a disk could be used. When the technology advanced so that both sides of a disk could be used for data, the old concept of sectors per track stuck; we still use the sectors-per-track measure as if a track could only have one side. That is why you can say there are 40 tracks on a diskette at nine sectors per track, and come up with a total of 720 sectors on a diskette.

The picture changes when we are talking about hard disks. Because hard disks often have more than one platter, the track-as-concentric-circle idea gives way to the image of concentric cylinders of data. A cylinder naturally describes all the same-radius concentric circles on which data can be recorded.

The Sector Layout for a Hard Disk

The 20-megabyte hard disk supplied with the IBM PC AT is a typical example of a hard disk. It has two rotating disks (platters) within it. Data is recorded on both sides of each platter, giving four tracks (heads) per cylinder, and head numbering from 0 to 3. The information that follows is for the IBM AT 20MB hard disk, and will vary for other manufacturers' hard disks. Let's look at some facts:

- A sector is 512 bytes.
- The 20MB hard disk has 17 sectors per track.
- A track on a hard disk contains (17 × 512 =) 8704 bytes (8.5K).
- There are four tracks per cylinder.
- The 20MB hard disk has 615 cylinders, giving a total of 2460 tracks.
- Each of the cylinders has 68 sectors (34,816 bytes per cylinder), for a total of 21,411,840 bytes.

The total-available-space figure can be misleading, for two reasons. First, a hard disk can be divided into four partitions, so that four separate, independent operating systems can coexist on a disk. The sizes of these partitions (the number of cylinders) are defined by the user via the FDISK program. On a hard disk you have control over the amount of space available to MS-DOS. Second, one whole cylinder and four sectors of each of the other cylinders are reserved, so the capacity is reduced to 614 cylinders of 64 sectors each, for a total of 20,119,552 bytes available for your programs and data. That is still a lot of space for data.

DISK SECTOR NUMBERING

Sectors are numbered from 1 to 9 (or 8, or 15) within each track, and two different numbering schemes are used within the MS-DOS manuals for numbering the sectors beyond the individual tracks. Logical (or absolute) sector numbers refer to all the sectors on the disk, and relative sector numbers refer to those sectors after the file allocation tables.

Let's now look at the numbering scheme that covers all the sectors on a disk. MS-DOS calls this form of addressing the *logical sector number* or *absolute sector number.*

Logical Sector Numbering

This numbering scheme allocates addresses to sectors beginning at the very first sector on a disk. All the sectors on a track (or cylinder) are assigned numbers before moving on to the next track. This is simple. However, if you deal with older disks that were created under version 1, it is useful to compare the association of logical sector numbers with physical sectors between version 1 and version 2. We will use as our example a double-sided diskette formatted for eight sectors per track.

Track 0, head 0 has logical sector numbers 0–7. In version 1, the sectors accessed by the second head on that track (track 0, head 1) are numbered 320–327. In version 2 they are numbered 8–15. The consequences of this difference in logical sector numbering are usually hidden from the user, but sometimes you need to know that the difference exists.

Hard Disk Logical Sector Numbering

With logical sector numbering on a hard disk, all the tracks on one cylinder are numbered before moving on to the next cylinder. The sectors read by head 0 are numbered 0–16, those read by head 1 are numbered 17–33, those read by head 2 are numbered 34–50, and those read by head 3 are numbered 51–67. The second cylinder begins at logical sector 68.

Relative Sector Numbering

One clear use of the term *relative sector* is in the hard disk partition table. The number of sectors that precede a partition is the relative sector of the beginning of the partition. However, other uses of relative sector exist. Relative sector numbering is sometimes associated with the allocation of space to files. Here the sectors are numbered starting with the sixth sector on the disk. (The first five sectors don't have sector numbers in this scheme.)

The name *relative sector* is not applied with uniformity throughout the IBM manuals. Relative sector is used in DEBUG's Load command description to describe what is otherwise called the logical sector.

Sectors within a Track

Now on to numbering the sectors within a track. If you need to compute addresses to find a location (not an unusual requirement in computer programming), it makes life simpler to have those addresses start at zero instead of one. According to this principle, the sectors within a track would be numbered from 0 to 8 for each track. MS-DOS, however, numbers the sectors from 1 to 9. The ROM BIOS disk routines also use 1 to 9; perhaps someone at Microsoft followed an offbeat but historical precedent in adopting this odd way of numbering sectors within a track.

DISK MAPS, OR WHAT'S WHERE ON AN MS-DOS DISK

The first few tracks of a diskette, disk volume, or disk partition have a highly predictable content. This section is devoted to showing you that content. After the first few tracks, we know the rest of the disk will either be unused or populated here and there with bits and pieces (clusters) of the files listed in the directories. To find out how and why files are mapped into disk sectors, read the "Allocation of Disk Space to an MS-DOS File" section later in this chapter.

For all versions of MS-DOS, the first few files are in the same sequence; they occur in the order shown in Figure 3.1.

The bootstrap, two file allocation tables, and the directory are not files addressable via the MS-DOS directory. If the disk is formatted with the /S option, three system files are added to it. They are, in sequence: IBMBIO.COM, IBMDOS.COM (or IO.SYS, MSDOS.SYS), and COMMAND.COM. The lengths of these files depend on the manufacturer and

File	Lengths:	SS,8	DS,8	SS,9	DS,9
Bootstrap sector		1	1	1	1
File allocation table—first copy		1	1	2	2
File allocation table—second copy		1	1	2	2
Directory		4	7	4	7

Figure 3.1: *The First Few Files on a Formatted Diskette*

the MS-DOS version number. IBMBIO.COM (and its namesakes) vary from 4 to 18 sectors; IBMDOS.COM (and its namesakes) are 34 sectors in version 2, and 56 sectors in COMPAQ's version 3.1A. In version 2 COMMAND-.COM occupies about 36 sectors, and in version 3, 46 sectors.

What's Where on a Hard Disk

As we have already seen, a hard disk can be divided into four partitions, so that four different operating systems and their files can exist on the same disk. A hard disk can also be divided into several different volumes. This means that the start of a volume or partition may not be at the beginning of the hard disk. However, the sequence of the files at the beginning of the volume or MS-DOS partition is the same as for the diskettes. Note that the FAT for disks of greater than 10MB will occupy more sectors in version 3 than in version 2.

DISK CAPACITIES

Using 360K to describe the capacity of a nine-sector-per-track, double-sided 40-track disk is not exactly accurate, but it is reasonable. It matches the use of 64K for 65,536 bytes of memory. There is overhead associated with storing data on a disk (the directory, file allocation table, etc.), so the whole 368,640 bytes of a 360K disk are never all available for your data. When referring to disk capacities, you have the choice of being accurate (which is not often necessary) or being approximate (using 360K) and remembering that you are being approximate.

You can use CHKDSK on a freshly formatted disk to show the sum of the space occupied by your version of the MS-DOS system (hidden) files, and the size of your COMMAND.COM. For those who are interested in accurate information, here it is for the TI Professional (version 1.13), and the COMPAQ (version 3.1A). Figure 3.2 summarizes the capacities described in the next few paragraphs.

Double-Sided Disk Capacity

The following figures are for a double-sided disk that has been formatted for nine sectors per track:

- A nine-sector, double-sided disk has room for 368,640 bytes of information. The boot record, the two file allocation tables (each FAT is two sectors long on nine-sector disks), and the seven-sector directory occupy 6144 bytes (twelve sectors). The available

remaining space is 362,496 bytes, which is the figure you see when you run CHKDSK.

- Where the disk contains the COMPAQ version 3.1A of MS-DOS and was formatted with the /S option, the system files (IBMBIO.COM, IBMDOS.COM, and COMMAND.COM) occupy another 122 sectors, or 62,464 bytes.
- If the double-sided disk is to be used purely for files of data (no programs), in version 2 you lose about 1.5% of the space (6144 bytes) to overhead and end up with 362,496 bytes available for data.
- If the disk contains COMPAQ MS-DOS version 3.1A, just over 17% of the space (62,464 bytes) is consumed by overhead, leaving 300,032 bytes for programs and data. Other MS-DOS capacities are similar.

Double-Sided, High-Capacity Disks

These figures are for a fifteen-sector-per-track, high-capacity disk for use with version 3 of MS-DOS:

- A high-capacity disk (fifteen sectors, double-sided) has room for 1,228,800 bytes of information (1.2MB). The boot record, the two file allocation tables (each FAT is 7 sectors), and the 14-sector directory occupy 14,848 bytes (twelve sectors). The available remaining space is 1,213,952 bytes, which is the figure you see when you run CHKDSK.

Single sided/ Double sided	Sectors /Track	Version	Bytes Available (Disk formatted without /S option)	Bytes Available (Disk formatted with /S option)
SS	8	TI 1.13	160,256	144,896
DS	8	TI 1.1	322,560	307,200
SS	9	COMPAQ 3.1A	179,712	118,272
SS	8	COMPAQ 3.1A	160,256	98,816
DS	9	COMPAQ 3.1A	362,496	300,032
DS	8	COMPAQ 3.1A	322,560	260,096
DS	15	COMPAQ 3.1A	1,213,952	1,152,512

Figure 3.2: *A Summary of Disk Capacities*

ALLOCATION OF DISK SPACE TO AN MS-DOS FILE

Disk space is allocated to a newly created or expanding file under control of the file allocation routines. These routines use the file allocation table (FAT) as a map of the occupied and available *clusters* on a diskette.

A cluster on a 40-track [PCCs] single-sided diskette is a single sector, 512 bytes. A cluster on a 40-track [PCCs] double-sided diskette is a unit of two contiguous sectors, 1024 bytes. The size of a cluster on a fixed disk depends on the size of the space allocated to DOS when the disk is formatted. Function 36H can be used to find the cluster size in sectors and the number of bytes in a sector. (Function 36H is an improved version of function 1BH.)

On double-sided disks, the clusters occupy both sides on one track before moving on to the next track. Any file greater than four clusters (eight sectors) on a 40-track [PCCs] disk will occupy parts of both sides of a double-sided diskette.

The file allocation table contains an entry for each of the allocable clusters on the diskette. Three hex digits are sufficient to address 4096 clusters, far more than are needed for the available space on a 40-track diskette in MS-DOS version 1. A brief computation—320 × 1.5 bytes = 480 bytes—shows that less than 512 bytes will suffice for the mapping of all possible clusters on a single- or double-sided, 40-track, eight-sector diskette. For convenience of access, a full sector is used for each copy of the file allocation table. The two FATs are found at sectors one and two, where sector zero is the first sector on the diskette.

In version 2, the FAT has to record the allocation of 360K bytes of diskette space, and megabytes of hard disk space. For nine-sector-per-track, 40-track diskettes, the FATs are each two sectors long. The size of the fixed disk's FATs is established when the FDISK command defines the area of the disk that will be devoted to MS-DOS.

The file allocation routines use the FAT to keep track of the chain of clusters that make up a file. The address of the first cluster is kept in the directory entry for that file (at bytes 26–27). All file-access routines use this as an index into the FAT to find the address of the next cluster, where in turn the address of the next cluster is found, and so on. The chain is terminated by a cluster value of FFx, where 8 <= x <= F. The chain of clusters in the FAT is lengthened or shortened to match changes in the file size as they occur. When a file is shortened and clusters become free, their locations in the FAT are set to zero. The arrangement of the three hex digits of the cluster address in the FAT is shown below. Within 3 bytes, L and H show the low- and high-order cluster addresses.

L	L	H	L	H	H
1	2	3	4	5	6

The allocation routines mark the position of unreadable clusters on the disk by FF7 entries in the FAT. Given the file allocation table scheme for keeping track of the contents of a file on disk, if the FAT goes bad, the entire contents of the disk become unavailable. For security two copies of the FAT are kept on the disk.

The first byte of the FAT describes the disk type. The first few sectors of each diskette are not allocable clusters (they are the boot, the two FATs, then the directory sectors), so the first two FAT entries are free for other purposes. Figure 3.3 shows the correspondence between the first byte of the FAT and the disk type for 5¼" disks. The second and third bytes are always FFFF.

DISK ACCESS MECHANISMS

MS-DOS has available absolute disk-access routines in addition to those related to file structure access (described in Chapter 2). This disk-access capability provides, within MS-DOS, alternatives to any ROM BIOS disk I/O routines. As usual, to enhance the portability of your MS-DOS programs, instead of using the disk-access routines in ROM BIOS, we recommend that you use the MS-DOS disk routines.

The relevant interrupts are as follows:

25H	Absolute disk read
26H	Absolute disk write

These routines read or write any number (specified by CX) of consecutive 512-byte sectors, starting at the sector number found in DX. The buffer into

Byte	Disk Type
FF	Double sided, eight sectors per track, 40 tracks
FE	Single sided, eight sectors per track, 40 tracks
FD	Double sided, nine sectors per track, 40 tracks
FC	Single sided, nine sectors per track, 40 tracks
FB	Double sided, eight sectors per track, 80 tracks
F9	Double sided, fifteen sectors per track, 80 tracks [PCCs]
F9	Double sided, nine sectors per track, 80 tracks
F8	Hard disk

Figure 3.3: *Disk Type Indicated by First FAT Byte*

which the data is read, or from which it is to be written, is specified in DS:BX; the drive number (0 = A, 1 = B, etc.) is specified in AL.

On return, the carry flag (CF) will be zero if the transfer was successful. If CF is 1, an error condition was detected. To unravel the cause of the error condition, see the section "Disk Access Error Codes" that follows (for the IBM PC and its compatibles, look under "Error Codes Specific to Interrupts 25H and 26H [PCCs]"). Use the code returned in AL to reference "Critical Error Handler Error Codes," and the code returned in AH to reference "Error Codes Specific to Interrupts 25H and 26H."

Note that whether or not a disk transfer is successful, on return, the stack will have grown by one word, because the interrupt call saved the original flags before executing your disk operation. Be sure to use a pop instruction to remove that word from the stack whether or not you want its contents. All the registers, except the segment register, are destroyed by this call.

Disk Access Error Codes

Two sets of codes exist for reporting disk errors. One set is common to the critical (or fatal) error handler (interrupt 24H) and the absolute disk read and write routines (interrupts 25H and 26H). The second set is specific to the absolute disk read and write routines; it is available in the IBM PC and its compatibles.

Critical Error Handler Error Codes—DI Register

These error codes are found in the lower half of DI.

Error	Meaning
0	Attempt to write on write-protected disk
1	Unknown unit [PCCs]
2	Drive not ready
3	Unknown command [PCCs]
4	Data error
5	Bad request structure length [PCCs]
6	Seek error
7	Unknown media type
8	Sector not found
9	(Printer out of paper)
A	Write fault (disk format error)
B	Read fault
C	General failure

Critical Error Handler Error Codes—AH Register

Hard errors on disk are indicated by bit 7 = 0 in AH. The low-order three bits of AH indicate whether a read or write was in operation, and the area of the disk affected.

Error	Area Affected	Read or Write
0	System files	Read
1	System files	Write
2	File allocation table	Read
3	File allocation table	Write
4	Directory	Read
5	Directory	Write
6	Data area	Read
7	Data area	Write

Error Codes Specific to Interrupts 25H and 26H [PCCs]

These errors are reported in register AH by the absolute disk read and write functions, 25H and 26H:

Error	Meaning
80H	Attachment failed to respond
40H	Seek operation failure
20H	Controller failure
10H	Bad CRC on disk read
08H	DMA overrun on operation
04H	Requested sector not found
03H	Write attempt on write-protected disk
02H	Address mark not found
00H	Error other than those above

PROGRAMMING FOR NONSTANDARD DISK DRIVES

You may want to add nonstandard disk drives or hard disks to your computer. Nonstandard in this context means a drive that is not already accommodated by MS-DOS. For example, a standard drive to the Zenith Z-100 system may be an 8-inch drive of 26 sectors per track, and the Zenith Z-DOS can use such a drive without trouble.

Nonstandard disk drives are easy to install because MS-DOS allows for installable device drivers. If the manufacturer of the disk drive supplies device-driver software that is compatible with MS-DOS, all you have to do is add the device-driver file to your working system disk and include the

name of it in your CONFIG.SYS file. If the name of the device driver is HIDISK.SYS, add to your CONFIG.SYS file the line

```
DEVICE = HIDISK.SYS
```

If the disk-drive manufacturer does not supply a device driver, you can write your own. You will need good documentation for the disk controller. *The IBM Technical Reference* for its Personal Computer contains a listing of the disk access code resident in the PC (or XT for hard disks) ROM.

If you are an inexperienced programmer, it will probably be only three or four months before your device driver achieves fault-free operation. Adding nonstandard drives can be done; the amount of work it takes to add a drive or drives varies according to how unlike the standard drives your drives are. What follows will serve as the barest minimum groundwork required to understand the task.

Alternate Disk Drives

The COMPAQ's ROM BIOS contains a set of parameters appropriate to the 5¼-inch disk drives it uses. The ROM BIOS parameters are found in a table pointed to by interrupt vector 1EH. This vector may be pointed at another, similarly structured table to accommodate a (not very) different disk drive.

The disk I/O interrupt vector 13H also points into the ROM BIOS, to the disk I/O routines. If you are willing to go to the trouble of writing your own disk I/O routines, you can add other drives. Your I/O routines will have to be compatible, in both calling sequence and result, with the ROM BIOS routines if they are to be used by MS-DOS. Where the characteristics of the disk drives are available to software in table form outside of IO.SYS (or its equivalent) or MSDOS.SYS, accommodating nonstandard drives is made easier.

Alternate Hard Disk Drives

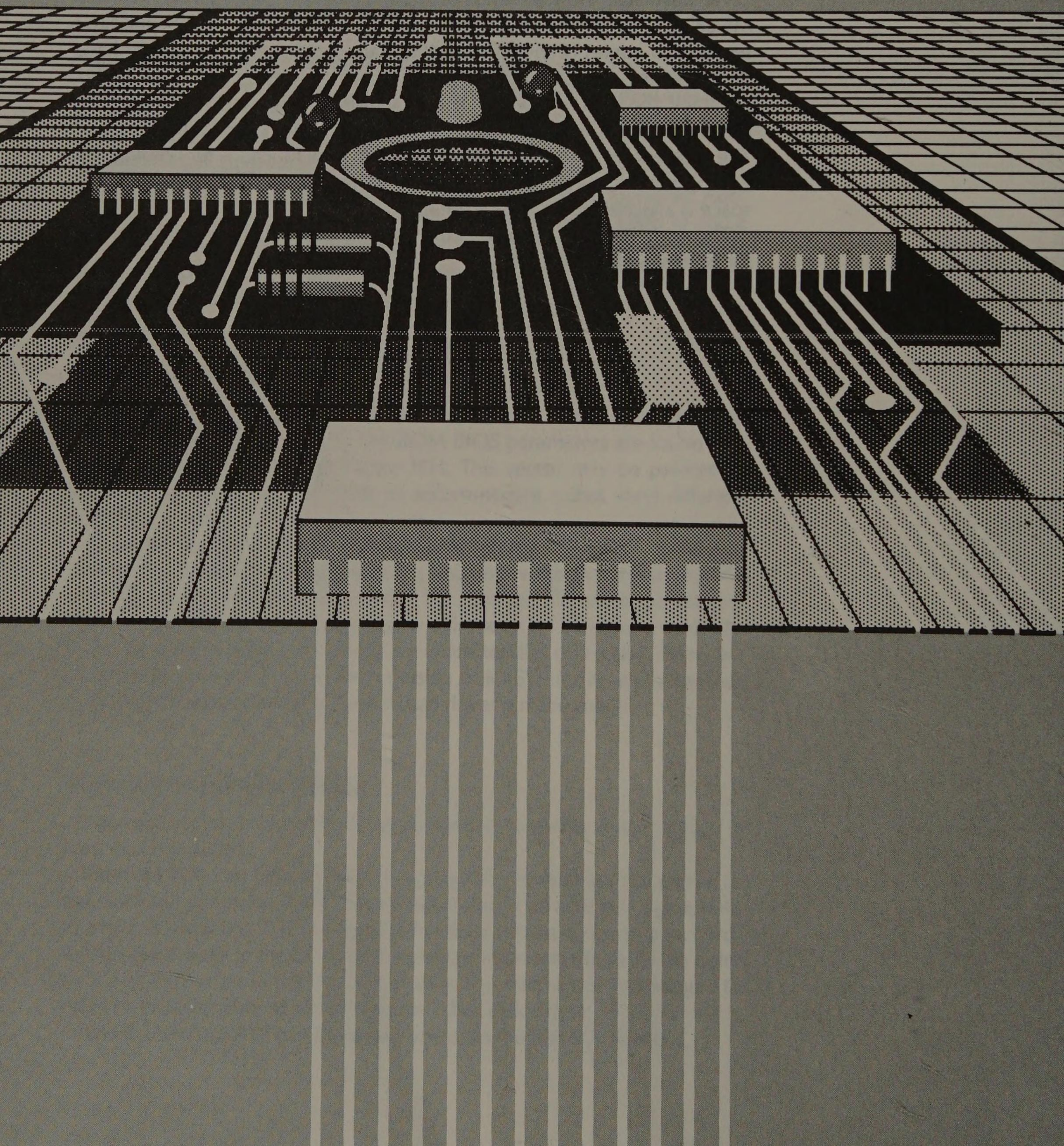
Interrupt vector 13H needs to be pointed to hard-disk driver code if a hard disk is installed.

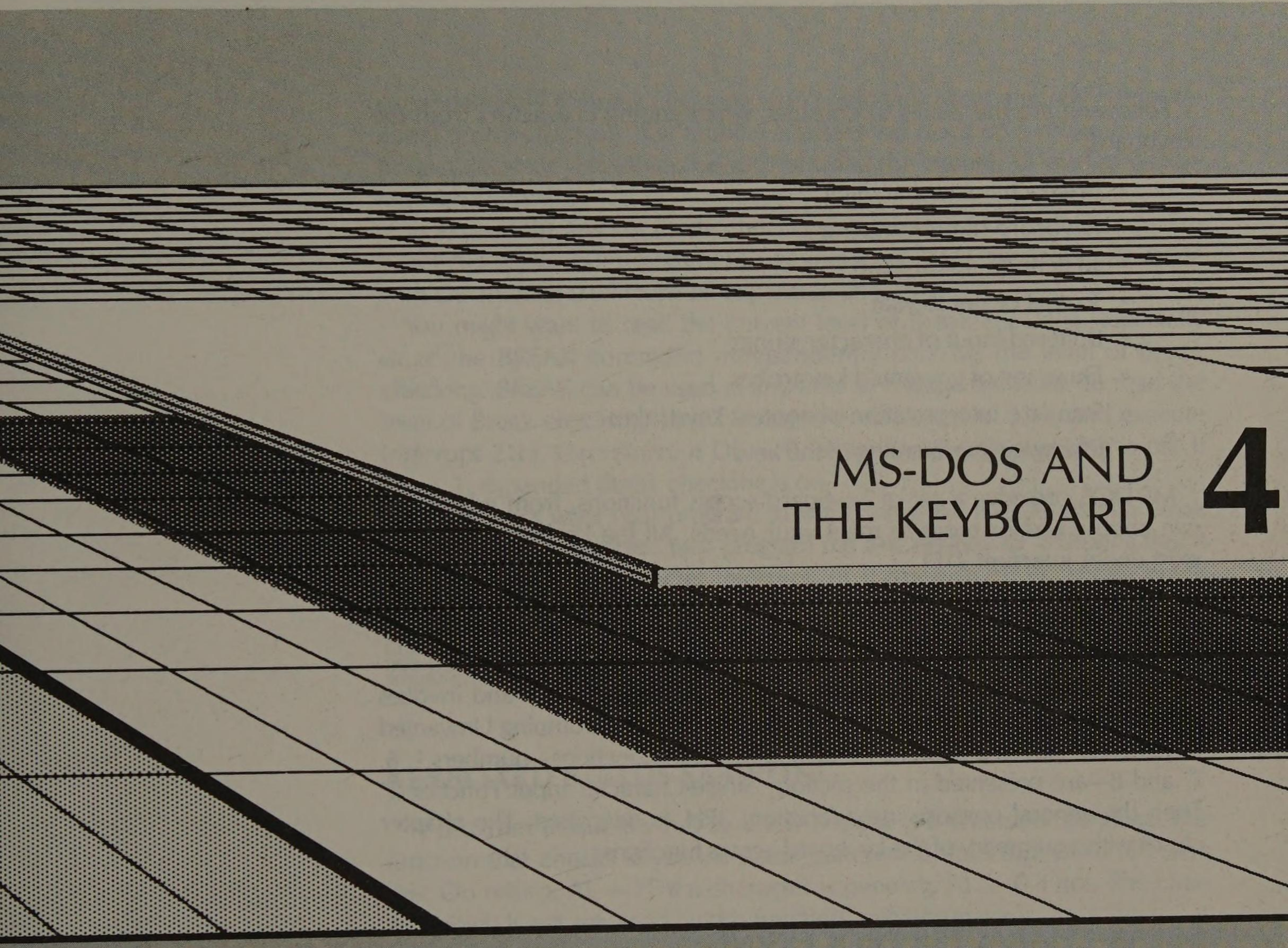
Interrupt 41H can be used as a vector to select one of several entries in the hard-disk parameter table. Each entry contains parameters appropriate to a type of hard disk. The hard-disk driver code adapts its behavior according to what it finds in the table. System initialization sets the vector to point to one of the entries in the drive-type table. This vector can be changed to point to another table of parameters for other disk drives. If a hard disk is installed, interrupt 40H can become a vector to the disk I/O routines.

SUMMARY

We have seen how all kinds of disks, as well as disk volumes and the MS-DOS partition in the hard disk, contain the same first few files: the boot sector, two copies of the file allocation table, and the directory. We have seen how the capacity of disks varies according to the number of sectors per track, the version of MS-DOS used to format them, and whether or not the /S option was used to transfer the system files when they were formatted. The different sector-numbering schemes, and the functions for reading and writing sectors, have been reviewed, as has the method by which disk space is allocated to an MS-DOS file.

The next chapter is devoted to the set of functions that MS-DOS provides for accessing the keyboard.





MS-DOS AND THE KEYBOARD 4

INTRODUCTION

MS-DOS provides a rich set of functions that a program can use to access characters entered at the keyboard. These functions permit the construction of a flexible and friendly interface with the user of your program. The somewhat formidable task of creating your own command processor (COMMAND.COM) is made simpler by the variety of MS-DOS keyboard input functions available.

Following are the issues to consider when getting characters from the keyboard:

- Break-character handling
- Keyboard-status sampling
- To echo or not to echo
- To wait or not to wait
- Buffered input of character strings
- Dumping of unwanted keystrokes
- Standard interpretation of nontext keystrokes
- Changing the meaning of the keys

MS-DOS provides a set of keyboard-access functions, from which you can choose one or more to meet your needs. All the keyboard functions exist under interrupt 21H.

Features of the relevant keyboard functions are included in the following discussions of each issue. The keyboard status sampling and buffered input functions (numbers B and A, respectively) are described in the so-titled sections. Function number C, which clears the keyboard buffer and invokes another keyboard function, is described in the section "Dumping Unwanted Keystrokes." All but one of the remaining keyboard functions—numbers 1, 6, 7, and 8—are presented in the section "Single-Character Input Functions." Then the general purpose input-function, 3FH, is described. The chapter closes with a summary of the keyboard-access functions.

BREAK-CHARACTER PROCESSING

The Break character is entered at the keyboard when the user wants to stop whatever program is executing. The Break character's key combination is Ctrl-Break [PCCs] or Ctrl-C. The standard action for MS-DOS to take when the Break character is entered at the keyboard is to execute interrupt 23H. Interrupt 23H is part of MS-DOS; it points to the code for processing the "stop-everything-right-now" command.

Normally, MS-DOS checks for the Break character being entered at the keyboard only during MS-DOS functions that access the keyboard, the screen, the printer, or the auxiliary devices (serial ports). However, you can expand that checking to include all MS-DOS function calls.

Function 33H—Expanded Break Checking

Function 33H and the BREAK command allow you to tell MS-DOS to check for the Break character every time any function call is used. This

expanded level of Break checking won't help you abort a purely computational process, but any disk and file accesses will open up a program to the Break character. (However, beware of using the expanded level of checking with DEBUG.)

To expand the level of Break checking, set AL equal to 1, set DL equal to 1, set AH equal to 33H, and execute interrupt 21H. The same call made with DL equal to 0 will turn off expanded checking.

You might want to read the current level of Break checking, especially since the BREAK command independently controls the level of Break checking. BREAK can be used at any time to change the level. To read the level of Break checking, set AL equal to 0, AH equal to 33H, and execute interrupt 21H. On return, if DL = 0, expanded Break checking is off. If DL = 1, expanded Break checking is on.

You may not want your program to lose control to MS-DOS when a Break character is encountered. Your program has two options for achieving this: it can trap the Break-character interrupt, or it can turn off expanded Break checking and use keyboard input functions that don't check for the Break character. The keyboard input functions that don't check for Break are the single-character input functions 6 and 7; all others check.

KEYBOARD STATUS SAMPLING

A program frequently has to know if there is a character pending. The function that checks keyboard status (function B, interrupt 21H) tells you this. On return, AL = FF if a character is pending; AL = 0 if not. The character itself is not returned by this function, only the status.

TO ECHO OR NOT TO ECHO

There are times when it is inappropriate to echo to the screen the character entered at the keyboard. MS-DOS provides for these times when it is operating in single-character mode; it gives you three function calls (6, 7, and 8) that do not echo the character entered at the keyboard.

Functions 7 and 8 are the same, except for their handling of the Break character. They both wait for a character to be typed at the keyboard. When one is ready, they return the character in AL without echoing it to the screen. A character is ready when it has been typed but not retrieved via function calls. When functions 7 and 8 are called and a character is ready, they return it in AL immediately.

Function 6 also does not echo a character entered at the keyboard. This function is described at the end of this chapter.

TO WAIT OR NOT TO WAIT

The question is whether your program, when accessing the keyboard, is willing to be suspended (to wait) until a character is entered at the keyboard.

Let's consider two different programs. At a certain stage of its operation, the first program has nothing else to do but wait for the next character to be entered at the keyboard. A more complex program may always have work to do, errands to run, during the rather long interval while it's waiting for input from the human being, who relies on the relatively slow mechanical activation of several pounds of flesh to communicate. (All right, a finger only weighs a few ounces.) The first program can afford to wait; the more complex program cannot.

The choice of waiting or not waiting applies only to single-character input, functions 1, 6, 7, and 8. The program using buffered keyboard input (function A) always waits.

To Wait

The first program is prepared to wait for the next character, and uses functions 1, 7, or 8 for single-character input. These functions wait for keyboard input (unless the character is ready, in which case they have no reason to wait). Functions 7 and 8 do not echo the character to the screen, and function 7 does not check the Break character.

Set AH to 1, 7, or 8, and execute interrupt 21H to wait for single-character input. If a character is pending, the functions return the character immediately in AL. If no character is pending, the functions wait until one is entered at the keyboard, then return it in AL.

Not to Wait

The more complex program needs to know whether a character is pending. To read the keyboard it can use either function 6, direct console I/O (with DL set equal to FF), or function B, keyboard status check, in combination with other keyboard input functions.

Function 6 returns immediately—with a character if one was ready, or without one if not. The more complex program can check the zero flag on return from function 6 to see whether it has time to do other nonkeyboard work. (See the end of this chapter for the full description of function 6.)

Function B, keyboard status check, tells the program when a character is pending. After that, one of the functions 1, 6, 7, and 8 can be called to get the character. It can be seen that the author of the more complex program has quite a variety of options to choose from.

In summary, if you want a function that will return with at least one character, use functions 1, 7, 8, or A. If your program has better things to do than wait for keyboard input, use function 6 alone, or the combination of function B (for status checking) and then your choice of functions 1, 6, 7, and 8.

BUFFERED INPUT OF CHARACTER STRINGS

Buffered keyboard input is provided by function A under interrupt 21H. This function is the heart of the software used to support the MS-DOS editing keys and the editing of single lines in EDLIN. Keyboard access via this function lets the user play with all the MS-DOS in-line editing capabilities before pressing the Return key.

To receive an input string terminated by a carriage return, your program will correctly set up an input buffer, and then call function A. This function accepts characters until the Return key is pressed. (A correctly set up input buffer is one where the first byte contains *n* and the buffer is of length *n* + 2.)

The user types characters, correcting (editing) as necessary, then finally types a carriage return. The function returns to your program with the second byte of the buffer indicating the number of characters entered, excluding the carriage return. The last of the *n* (or fewer) characters is always 0DH, the carriage return.

As when the user is entering or editing a line within EDLIN, if there is no more room in the buffer for any character other than the carriage return, a bell sounds when another character is typed, and that character is ignored.

To call the buffered keyboard input function, point DS:DX to the prepared buffer, set AH equal to A, and execute interrupt 21H.

DUMPING UNWANTED KEYSTROKES

It is sometimes desirable in a program to accept input from the keyboard only after disposing of characters that might be in the input stream for other reasons. Function C (which clears the keyboard input buffer and executes an input function) has been provided for this purpose. Function C is an envelope function to the other keyboard functions 1, 6, 7, 8, and A; your choice of one of them is called by function C after the input buffer has been cleared.

To clear the keyboard input buffer and invoke a keyboard input function, set the desired input function number in AL, then set AH to C; set DS:DX if

you are calling function A; set DL if you are calling function 6; and execute interrupt 21H.

For example, if your program is to be used by children who are new to computers, there is a chance they will be typing on the keyboard long before your program is ready to read what they type. If the program asks for the user's name, the sequence of operations should be as follows:

1. Display the question "What is your name?"
2. Clear the keyboard input buffer.
3. Call the buffered input function.

Your program would probably use function 9 (string output) to display the question, then take the shortcut offered by function C, calling it with A (the buffered input function's number) in the AH register.

CHANGING THE MEANING OF THE KEYS

This section gives an overview of the processing that occurs during the time that your finger depresses a key on the keyboard and a character (or characters) appears on the display. (Note: The computer used in the example is one of the IBM PC compatibles, the COMPAQ. Most other computers with MS-DOS operate similarly.) This material may take a little time to absorb; at the end of this section you may find that you know more about the keyboard and character translation than you ever thought you wanted to know.

Before examining a map of our journey, please note that if you plan to use the key-reassignment feature made available in version 2 and beyond of MS-DOS, you need only read that particular section and learn the difference between scan codes and extended ASCII codes.

We will first examine how the keys are arranged on the keyboard, how the keys are numbered, and what scan codes are. While on the subject of codes, we will look at the difference between scan codes, extended codes, and ASCII codes. Next we will examine the different software components that are brought into play when a key is depressed.

To see where we might make changes in order to alter the meaning of a key, we will follow the path taken by the keystroke codes when the first function key (F1) is pressed, noting where diversions and reconstruction might affect changes in the meaning of the F1 key along the way. After that, we will show how the key-reassignment feature makes it a simple task to change the meaning of a key. Let's begin with the keyboard and its scan codes.

The Keyboard Layout

A glance at the COMPAQ DESKPRO keyboard reveals three major groups of keys: (1) the (almost standard) typewriter keyboard (in the middle); (2) the ten function keys (on the left); and (3) a group of keys serving two purposes, numeric data input and cursor control (on the right). Other MS-DOS computers may shift the function keys from the left side to a row at the top, and some have only cursor-control keys without the additional 10-key keypad for numeric data entry.

The keys on the COMPAQ DESKPRO are numbered (starting at 1, not 0) from left to right and top to bottom, within the three key groups (typewriter, function, and numeric/cursor). This number, which corresponds to the position of the key on the keyboard, is called the *scan code,* or sometimes the *key code,* reasonably enough.

The Keyboard's Scan Codes

A diagram of the keyboard showing the scan code for each key can sometimes be found in an appendix of the BASIC release 3.0 manual. In case you do not have that diagram, here are the scan codes for each keyboard group in the COMPAQ. Feel free to skip to the "ASCII and Extended ASCII Codes" section if you already know, or can find, your keyboard's codes.

The Typewriter Group's Scan Codes

The typewriter group of keys is the rectangle formed by the Esc key at top left, the backspace key at top right, the Alt key at bottom left, and the CapsLock key at bottom right. The rectangle includes the PrtSc key, but its perfection is marred by the intrusion of the left half of the Ins key, which belongs in the third key group.

The Esc key has a scan code of 1. The left-to-right, top-to-bottom numbering scheme gives the backspace key a scan code of 14, the Tab key a scan code of 15, the return (Enter) key a scan code of 28, the Ctrl key a code of 29, and so on, up to the CapsLock key's scan code, which is 58.

The Function Keys' Scan Codes

The function keys are numbered next, again from left to right and top to bottom. The F1 key's scan code is 59, F2 is 60, and F10 is 68.

These are the keys whose meaning we are most likely to want to change. The ten function keys can mean anything the currently running program chooses. Their presence enables software developers to provide users with single keystrokes that can save a lot of time and confusion. You have probably seen how BASIC uses the F1, F2, and F3 keys to mean LIST, RUN, and

LOAD, and how EDLIN and MS-DOS at the prompt level use those same keys for in-line editing functions.

The Numeric/Cursor-Control Group's Scan Codes

Finally, the remaining right-hand keys are numbered. The scan code for the NumLock key is 69; for the plus-sign key 78; and for the Del key (the last code on the keyboard), 83.

Now that we know that the scan code is just a key position, let's see how this code, or something like it, is used in extended codes.

ASCII and Extended Codes

Several keystrokes and keystroke combinations, such as the function keys, the Home key [PCCs], and the PgUp and PgDn keys [PCCs], are not represented in the full ASCII 256-character set. To resolve the problem of how to tell a calling program that those keys have been pressed, another way of passing nonstandard keystrokes to the applications program has been devised. The concept of extended codes has been introduced to represent nonstandard keystrokes and keystroke combinations.

Extended codes are two-byte codes, where the first byte is null (zero). The second byte has no connection with ASCII. It is simply the scan code of the key pressed. Your BASIC manual may provide a match between the second byte of an extended code and the key that it represents.

This arrangement is a little strange, but it works reasonably well for some of the extended codes. For example, it is fairly convenient to represent the F1 key as 0,59, and the Home key as 0,71.

Let's look at the software components that work for us when our program gets a character from the keyboard.

From Fingertip to Applications Program

When you strike a key, the keyboard sends a scan code to the PC; the scan code is received by the computer's ROM BIOS keyboard interrupt routine. The ROM BIOS keyboard I/O routines convert the scan code into ASCII or extended code, and hand it out in a single word to the software seeking input from the keyboard via interrupt 16H. Often, it is MS-DOS that has called interrupt 16H. Let's review that process.

When characters are returned to an applications program that uses the MS-DOS keyboard input functions, the keystroke information has traveled from the keyboard to ROM BIOS via interrupt 9H, from ROM BIOS to MS-DOS via interrupt 16H, and from MS-DOS to the applications program via interrupt 21H. (Note: We are not seeing the full complexity here, just the essentials.)

Before we look at how we can change the meanings of keys, we need to look a little more closely at the path taken by information from the keyboard before it reaches our program, and how that information gets translated along the way. Once we know the path, we will see that there are several points at which changes can be made. Let's follow the path of the F1 keystroke from keyboard to applications program. We are still using the COMPAQ as our example.

The Path That F1 Takes

When the F1 key is pressed, scan code 59 is sent to the COMPAQ, generating a keyboard interrupt. Interrupt 9H points to the keyboard interrupt service routine in ROM BIOS. This routine translates the F1 key's scan code (59) into a two-byte extended code (0,59).

System-level programs (such as MS-DOS) that want to get the next character from the keyboard use interrupt 16H, which points to the keyboard I/O routines in ROM BIOS. These routines return the F1 keystroke as an extended code in the AX register, with AL = 0 and AH = 59. When applications-level programs (the type that you and I usually write) want a character from the keyboard, they call the MS-DOS keyboard input function (number 1, 6, 7, or 8) to get a character.

The MS-DOS routines return characters one at a time. One call to the function gets the character code, unless MS-DOS has an extended code to give out. MS-DOS tells its caller that it has an extended code to return to the caller by returning with AL = 0. When the applications program sees the zero code, it must call the keyboard input function one more time to get the extended code.

Now that we're familiar with the path that F1 takes, let's look at the possibilities for changing the meaning of the character along the way. The possibilities include altering the path, as well as intervening here and there.

Changing the Meaning of F1

We will look at three approaches for changing the meaning of the F1 key. The first takes much work; the second avoids tinkering with the system; the third does what we want most economically. Let's look at the "rewrite everything" approach.

The Big Rewrite

This approach involves the most work. Your applications program takes over all the functions supplied by the ROM BIOS interrupt routines, the ROM BIOS character translation routines, and the MS-DOS keyboard input

function calls. To use this approach, you must

1. write the substitute keyboard interrupt routine and point vector 9H to it;
2. write the substitute character-translation-and-fetch routine and point interrupt vector 16H to it; and
3. write a filtering routine that catches all keyboard input functions within interrupt 21H, passing on other function calls untouched, and point interrupt 21H to it.

All this work is not recommended unless you have a lot of time on your hands and are desperate to produce the quickest response possible to each character entered at the keyboard. Reviewing this wholesale rewrite approach has, however, revealed a few less laborious possibilities.

The Minimal-Intervention Approach

One approach that totally avoids tinkering within the system is to have your applications program do the character translation. For example, your applications program could have a central routine that

1. gets characters from the keyboard using the (usual) MS-DOS interrupt 21H keyboard input function calls;
2. compares those characters (and extended codes) against a table that you have included in the program, holding the characters or extended codes to be translated (the table also holds their translations); and
3. on finding a match, translates the character before passing it to the rest of the applications program.

This approach does not involve changes to any interrupt vectors.

If the translations between scan codes and ASCII or extended codes that ROM BIOS performs do not conflict with your needs, the cleanest and most economical way to provide the (additional) translation you need is to trap the calls to ROM BIOS that are made via interrupt 16H.

The Economical Approach

The most economical approach is to point interrupt vector 16H to your own routine, which would

1. accept the calls that normally go to the ROM BIOS keyboard I/O routines via interrupt 16H;
2. pass those calls on, unchanged, to the address in ROM BIOS that vector 16H normally points to (i.e., the value normally found in locations 58H–5BH);

3. look at, and where necessary translate, the characters or extended codes that the ROM BIOS routines return; and
4. pass those characters to whichever routine it was that thought it was calling the ROM BIOS I/O routines directly.

If your translation requirements involve substituting several characters for a single character or extended code, your routines would also have to be smart enough to pass the multiple character strings out to the caller, one by one.

Note that under this method, any software that uses vector 16H (not just MS-DOS) will have characters translated by your routine.

Under any of these methods, be sure that when your software terminates, it resets all the interrupt vectors it changed!

Adding routines and substituting interrupt vector contents is not prohibitively difficult, but using key reassignment may be a better approach for your needs.

Keyboard Key Reassignment

Key reassignment makes it much easier to change the meanings of various keystrokes. The reassignment capability costs a few bytes of memory (about 1.5K), but that is its only disadvantage. One or two implementations of MS-DOS do not contain the keyboard key reassignment capability described here; almost all do. Let's now discuss what to do to make key reassignments happen, then go through an example in which we change the meaning of one of the function keys (say, F10) so that a single keystroke at the MS-DOS prompt level produces a directory listing.

Changing the Meaning of a Key Using Key Reassignment

The meaning of a key is reassigned when a unique sequence of characters, known as a control sequence, is seen by keyboard functions 1, 2, 6, and 9 within interrupt 21H. Key reassignment operates only if the ANSI.SYS software is incorporated in MS-DOS. ANSI.SYS is a device driver for keyboard control. To incorporate it into MS-DOS, add the command

```
DEVICE = ANSI.SYS
```

to your configuration file. If you don't yet have that file, you can create it using COPY, as follows. Type

```
COPY CON CONFIG.SYS
DEVICE = ANSI.SYS
<Ctrl-Z>
```

If you are going to use COPY to create the configuration file, be careful that

CONFIG.SYS does not exist, or that you don't mind destroying it if it does. If in doubt, use EDLIN instead of COPY.

Once the CONFIG.SYS file exists, reboot MS-DOS to get the ANSI.SYS device driver included in the resident portion of MS-DOS.

We show two ways to reassign the meaning of an ASCII or extended code. Both involve sending out, via DOS function calls 1, 2, 6, or 9, the correct stream of control characters to effect the change in meaning desired. Sending a string to the display (function 9) is simplest. In the first example, we write a very small program to send out the characters. In the second, we use instead the MS-DOS PROMPT function to send the escape sequence. In both examples, the stream of control characters is as follows:

1. the Esc code, a byte value 1BH;
2. the left bracket character ([);
3. a string of ASCII characters that express in decimal the ASCII code for the character being mapped;
4. the semicolon character;
5. the decimal ASCII value of the character(s) that replace the one being translated;
6. the character p.

If you are reassigning the meaning of an extended code, or replacing an ASCII-representable character with an extended code, items 3 and 5 above use a string of characters that expresses in decimal the extended code. The two extended code values are separated by a semicolon.

Giving It a Whirl

You can experimentally change F10 to mean DIR by using DEBUG. Here's how:

Call DEBUG. Use the Assemble command to enter the code that sends a control sequence to the display. Type the following lines (exactly):

```
a 100
mov ah,9
mov dx,200
int 21
<return>

e 200 1b'[0;68;"dir";13p$'

g=100 107

q
```

Now you are back at the prompt. Press the F10 key.

What Happened in the Whirl

The three-instruction program at CS:100 calls function 9 within interrupt 21H. It sends to the display the string at DS:200 (DEBUG has been kind enough to set the segment registers for us; CS = DS).

The Enter instruction creates, at DS:200, the control sequence that reassigns the meaning of the F10 key. Let's examine that control sequence:

1b	is the code for Esc.
'	tells the Enter command to treat the following characters as a string.
0;68	is the extended code for F10.
;	separates the code from its new meaning.
"	tells the control-sequence processor that the next characters will replace the keystroke(s) being translated.
dir	is part of the translation.
;13	adds a carriage return to "dir".
p	is always the last character of the control sequence.
$	terminates the string that function 9 puts out.
'	terminates the string for the Enter command.

Giving It a Whirl Using the PROMPT Command

Instead of using DEBUG, we can use the PROMPT command to send the escape sequences. We can't just type them in at the keyboard, because MS-DOS interprets the Escape character to mean "wipe out the current input line and start again." That defeats entry of the Escape character; it gets interpreted before ANSI.SYS has a chance to see it. However, the PROMPT command accepts codes that specify how the prompt should appear. One of those codes is $e, which gets translated into the Escape character.

It may seem odd to use the PROMPT command as a vehicle for sending escape sequences. What does the DOS prompt look like when we are done? We can ignore that problem by using another feature of the PROMPT command. Entering PROMPT with no arguments resets the prompt message to the standard prompt for your machine (e.g., A>).

To change the F10 key to mean DIR, type

```
PROMPT $e[0;68;"DIR";13p
PROMPT
```

As you type this, you will notice the absence of the regular DOS prompt between lines one and two.

We have now seen how to use the built-in key-reassignment feature. The rest of the chapter describes the single-character input functions, ending with a summary of all the keyboard-access functions.

SINGLE-CHARACTER INPUT FUNCTIONS

We have mentioned the four MS-DOS single-character input functions here and there, but we haven't provided in one place a description of all their features. We'll do that now. Three of the input functions (1, 7, and 8), always get a single character from the keyboard (assuming there is no Break-character intervention); the other, function 6, returns immediately, even if no character is available. The functions differ in their Break-character detection and whether or not the character is echoed to the screen.

Functions 1 and 8—Single-Character Input with Ctrl Break Detection

These two functions are the same, except that function 8 does not echo the input character to the display. Both functions wait for a character to be typed at the keyboard (unless one is ready), then return it in AL.

To implement this function, set AH equal to 1 or 8 and execute interrupt 21H. On return, the character will be in AL.

If keys or key combinations are pressed that cannot be represented by a standard ASCII code, an extended code is returned. When this happens AL will contain 00, and another call must be made to the function to get the extended code.

Function 7—Single-Character Input without Ctrl Break Detection

Function 7 is a single-character input function with no echo. It is the same as function 8, except that no check is made for the Break character. The function waits for a character (unless one is ready), then returns it in AL.

To call the function, set AH to 7, and execute interrupt 21H: On return, the character is in AL; it has not been echoed to the screen.

Function 6—Single-Character Input without Wait, Alias Direct Console I/O

Function 6 is used either for input from the keyboard or for output to the screen, but not both at the same time. If DL = FF on entry, it is a keyboard function; otherwise, it is an output function, where the character in DL is output to the screen and no keyboard interaction takes place.

This is the only keyboard input function that, if there is no ready character, does not wait for a character to be typed. On return, the zero flag is a status indicator. If the zero flag is set, there was no character ready. If a character was ready, the zero flag is clear and the character is in AL. Remember that for function 6 to act as a keyboard input function, DL must be set to FF.

Function 3FH—Reading from the Standard Input

Function 3FH can be used to get a specified number of characters from a file or device. If that device is the standard input, function 3FH will read characters from the keyboard.

To read from the keyboard, with echo to the display and Ctrl-Break detection, put the number of characters to read in CX, the standard input device handle (0001) in BX, the address of the buffer for the characters in DS:DX, set AH to 3FH, and execute interrupt 21H.

The function will return only when the Return key is pressed, even if either less or more than the requested number of characters has been entered.

SUMMARY

The eight keyboard-access functions can be separated into the following groups: the five different keyboard input functions, one of which returns status information; a separate keyboard status checking function; an envelope function that calls one of five input functions after clearing the input buffer; and the general purpose input-function with standard input as the input device.

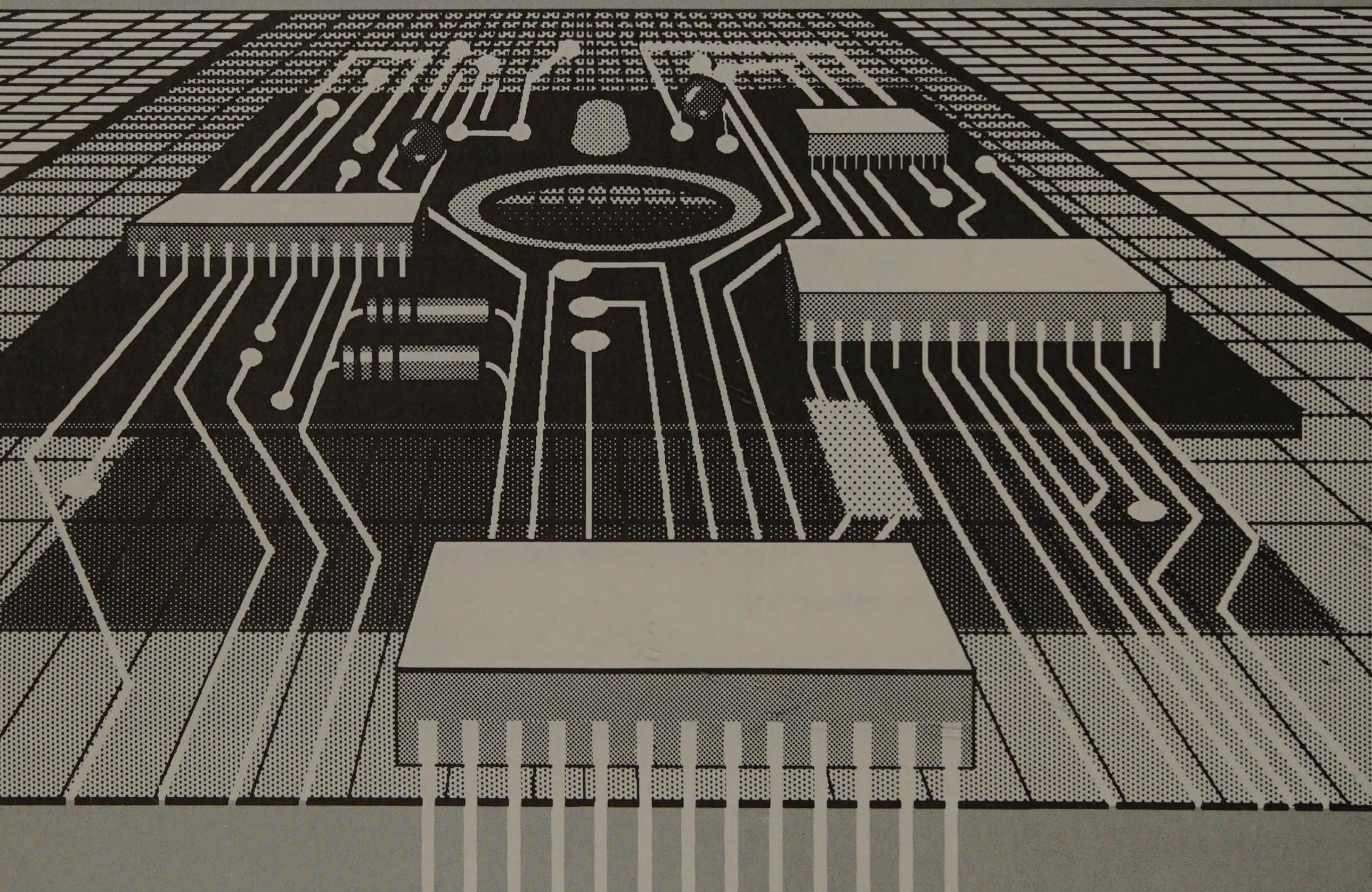
To invoke one of these functions, put the function number in register AH, set the arguments (if any) as necessary, and execute interrupt 21H. In the following summary (Figure 4.1), a function that "waits for" a character will return immediately if one is ready.

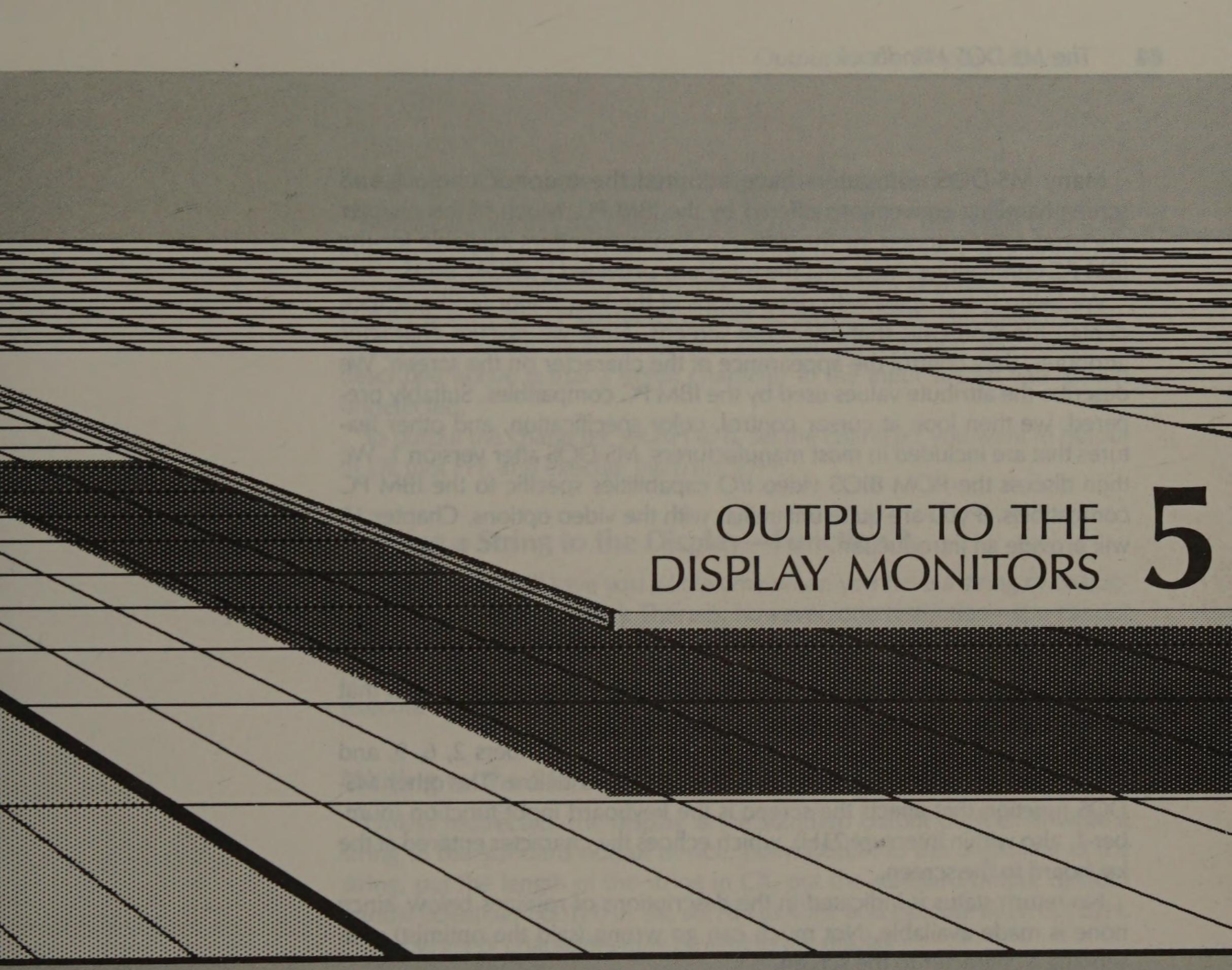
Function Number	Purpose and Arguments
1	Waits for and echoes a character, which is returned in AL; Break character is detected.
6	If DL = FF, returns with character in AL if one was ready, zero flag is set if none ready. No echo is performed, and Break character is not detected.

Figure 4.1: *Keyboard-Access Function Summary*

Function Number	Purpose and Arguments
7	Waits for a character, which is returned in AL. No echo is performed, and Break character is not detected.
8	Waits for a character, which is returned in AL. Same as function 1, except the character is not echoed. Break character is detected.
A	Waits for a string (which may be empty), terminated by the Enter key. Full MS-DOS in-line editing features are supported. String is echoed; Break is detected. On entry, DS:DX points to an input buffer.
B	Checks keyboard status. Does not return a character. If one is available, AL = FF; if no characters are ready, AL = 00.
C	Clears keyboard input buffer and invokes keyboard input function. On entry, AL specifies which of functions 1, 6, 7, 8, and A is to be invoked after clearing the buffer.
3F	Same as A, except that only the specified number of chararacters are returned in the buffer.

Figure 4.1: *Keyboard-Access Function Summary (continued)*





OUTPUT TO THE DISPLAY MONITORS 5

INTRODUCTION

This chapter is about using MS-DOS and some computers' ROM BIOS functions to output characters and graphic images to the screens. MS-DOS is not rich in display-access capabilities, nor does it offer a standard graphics interface package (standard graphics interfaces are not widely accepted, in part because it is hard to create an efficient and adequately comprehensive package, and in part because of differences in the various screens that have to be addressed).

Many MS-DOS computers have adopted the monitor choices and screen-handling conventions offered by the IBM PC. Much of this chapter discusses screen access for the different display monitors available for the IBM PC compatibles, including the functions provided by ROM BIOS.

We begin the chapter with descriptions of the four major (and one incidental) function calls that affect the screens. Next we discuss character attributes; they control the appearance of the character on the screen. We describe the attribute values used by the IBM PC compatibles. Suitably prepared, we then look at cursor control, color specification, and other features that are included in most manufacturers' MS-DOS after version 1. We then discuss the ROM BIOS video I/O capabilities specific to the IBM PC compatibles. If you are quite unfamiliar with the video options, Chapter 15 will provide an introduction.

FUNCTION CALLS THAT AFFECT THE SCREEN

MS-DOS provides functionally simple character-output routines that operate on characters or strings of characters.

The four function calls that output to the screen, numbers 2, 6, 9, and 40H, exist within interrupt 21H. They are described below. The other MS-DOS function that affects the screen is the keyboard input function (number 1, also within interrupt 21H), which echoes the character entered at the keyboard to the screen.

No return status is indicated in the descriptions of routines below, since none is made available. Not much can go wrong (said the optimist) with sending a character to the screen.

Sending a Character to the Display (with Backspace Translation)—Function 2

When you output a backspace (08H), this function moves the cursor one position to the left. The character at that position remains visible until another character is written to that position.

To output a character, set AH equal to 2 and execute interrupt 21H. The character in DL is output to the screen.

When you use this routine, there is a chance that control will be whipped away from you if a Break character is detected in the input stream. If you wish to retain control in this eventuality, before using this function you will need to redirect interrupt 23H to your own Break routine. To do this you can use function call 25H, in interrupt 21H, to point interrupt vector 23H to your own code. Don't forget to first use function call 35H, in interrupt 21H, to get the existing vector so you can restore it later.

Sending a Character to the Display—Function 6

Microsoft calls this the *Direct Console I/O* function. The single character in DL is output to the screen. No translation of the backspace or any other character takes place. Since this function becomes a keyboard input function if DL = FF, the FF character cannot be sent to the screen using this routine. The other significant difference between this function and number 2 described above is that a Break character in the input stream will remain undetected.

To output the character, set AH to 6, set the character you want to output in DL (not FF), and execute interrupt 21H.

Sending a String to the Display—Function 9

This function will save you a little time when you have a string of characters to output to the display. There is, however, one restriction—you cannot output a dollar sign. To use this function, point DS:DX to the beginning of a character string that is terminated by a dollar sign (24H), set AH to 9, and execute interrupt 21H.

Writing to the Standard Output—Function 40H

Unless redirected, the display is the standard output device. To write a string to the standard output device, point DS:DX to the beginning of the string, put the length of the string in CX, put the standard output device's handle (identifier 0001H) in BX, set AH to 40H, and execute interrupt 21H.

Any string of characters may be sent to the display.

CHARACTER ATTRIBUTES

Each character on the display can have one of several attributes. An *attribute* is a modifier to the basic character, controlling the way it appears on the screen. Each bit in the attribute byte controls a different feature of the character.

To demonstrate the use of character attributes, as well as the functions that are offered in ROM BIOS, we need to be aware of the two video display options offered by the IBM PC compatibles:

- a monochrome display driven by the monochrome adaptor, and
- color or black-and-white display (or television) driven by the color/graphics adaptor.

Other options are available, for example IBM's enhanced graphics adaptor, and the Hercules adaptor. These adaptors usually emulate the monochrome

and color/graphics adaptors and then provide more options.

We will see the useful values for monochrome attribute bytes. The effect of each of the eight bit positions in the attribute byte is described for the color/graphics adaptor. The attribute byte can have a slightly different effect on the character, depending on whether it is directed to the display via a monochrome display adaptor or a color/graphics adaptor operating in alphanumeric mode. However, two features of the attribute byte, blinking and high intensity, which are each represented by a single bit, remain consistent between monochrome and color displays.

When the display adaptor is in alphanumeric mode, each character's byte in the screen's memory is followed by its attribute byte. It takes two bytes to display a character in this mode. When the display adaptor is operated in the APA (all-points-addressable, i.e., graphics) mode, the meaning of an attribute byte evaporates. Each bit (or combination of bits) in the memory for the screen represents a point (or very small area, called a pixel) on the screen.

With MS-DOS function calls you can only output to "the display" (type not defined). You cannot specify an attribute byte different from the value adopted by the manufacturer. This value gives you what you normally see on your computer's screen—usually a white-on-black character.

Let's now look at how the value of the attribute byte affects the appearance of characters on the display driven by the monochrome adaptor.

The Effect of Character Attributes on the Monochrome Display

The attributes that are available on a display driven by the monochrome adaptor are as follows:

- Normal—white character on black ground
- Reverse video—black character on white ground
- Underline (for white characters only)
- Blinking character
- High-intensity character

As you will see, not all attribute combinations are available. For example, underlined reverse video cannot be obtained.

Figure 5.1 shows the purpose of the bits in the attribute byte as they are recognized by the monochrome adaptor. Blinking and high intensity are each caused by a specific bit in the attribute byte: bit 7 for blinking and bit 3 for high intensity. (The blinking bit is interpreted by the monochrome and color/graphics adaptor in the same way.) The other six bits have a limited range of effective values.

Figure 5.2 shows a table of useful attribute values and the effects they create. From this figure it can be seen that the adaptors assign a higher priority to the underline request than the reverse video request; it is for this reason that underlined reverse video is not available.

Character Attribute Values with the Color/Graphics Adaptor

The adaptors used by most MS-DOS computers operate in two modes: *text* (alias alphanumeric or A/N), and *graphics* (alias all-points-addressable or APA). The attribute byte gives full control of the character's foreground and background colors when the adaptor is operating in the text mode. (Note: For a description of color control in graphics mode, see Chapter 15.)

The attribute byte for colored characters is perhaps the easiest to understand. The colors are made from combinations of red, green, and blue. The eight bits and their functions appear in Figure 5.3.

Each character can have one of sixteen (defined by bits 0 to 3) foreground colors and any of eight (defined by bits 4 to 6) background colors. Each character can be made to blink by setting bit 7.

Now let's briefly look at the video options available with the IBM PC compatibles. Chapter 15 gives a fairly thorough view of the video options. We recommend you read it if you have unanswered questions at the end of this chapter.

	7	6	5	4	3	2	1	0
	FG	Background			Foreground			
Normal character	b	0	0	0	i	1	1	1
Reverse video	b	1	1	1	i	0	0	0
Underlined character	b	0	0	0	i	0	0	1
No display, black on black	b	0	0	0	i	0	0	0
No display, white on white	b	1	1	1	i	1	1	1

Bit 7 when set makes the character blink

Bit 3 when set makes the character high intensity

Figure 5.1: *Attribute Bytes Recognized by the Monochrome Adaptor*

Video Options

The IBM PC compatibles offer two basic types of monitor drivers.

One video option is a high-quality display (monitor) and a monochrome adaptor. This operates in one mode only: 80 × 25 characters, black and white. It has buffer space for a single page of display.

Attribute Value (Hex)	Effect
00	No display, black on black
01	Underlined character
07	Normal character, white on black (this effect is also obtained with attribute values 02 through 06)
08	No display
09	Underlined, high-intensity character
70	Reverse video
71	Underline only, not reverse video
78	Reverse video, high intensity; the character appears thinner, more finely drawn
79	High intensity, underlined, not reverse video
81	Blinking underlined character
87	Blinking normal character
89	Blinking, underlined, high-intensity character
F0	Blinking, reverse video
F8	Blinking, reverse video, high intensity; the character appears thinner

Figure 5.2: *Useful Monochrome Attribute Byte Values*

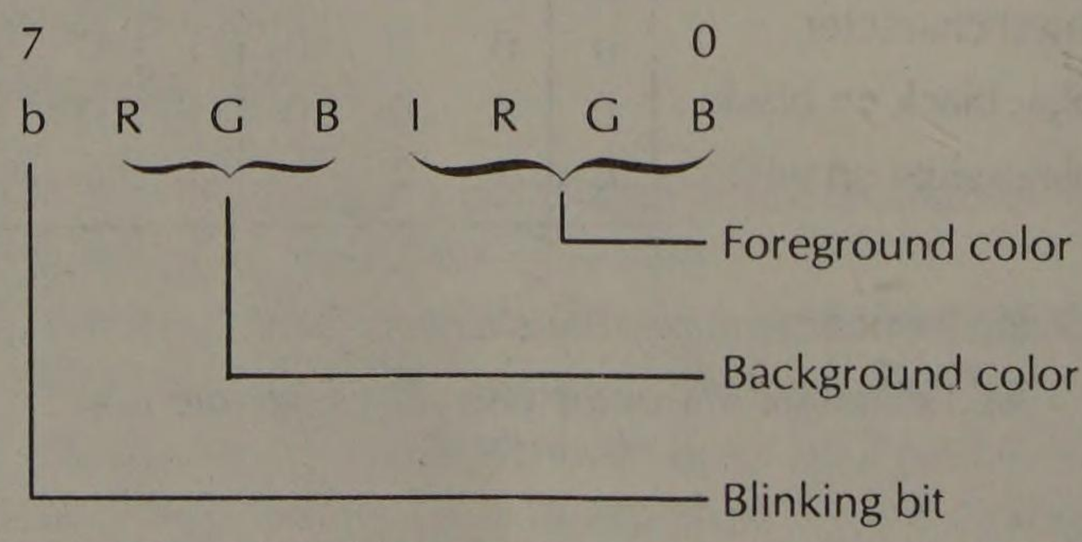


Figure 5.3: *The Attribute Byte for Color Selection*

The other option is the color/graphics adaptor, with a suitable monitor or television of your choice. This adaptor allows you to output both characters and single dots to the screen (a character is made from a matrix of 5 × 7 dots within an 8 × 8 box) and, as the name implies, allows you to control the colors of the characters and the dots. Noncompatibles may have different character matrices.

The color/graphics adaptor operates in several modes; it offers four text and two graphics modes. You have a choice of a 40 × 25 or an 80 × 25 character display (a 40 × 25 mode is required for most televisions and lower-resolution monitors, since it doubles the width of the character). Color is available in both of these character widths. In graphics, the choice is between two resolutions: 320 × 200 or 640 × 200. Color is available in the 320 × 200 mode; the 640 × 200 mode is black and white only.

When the color/graphics adaptor is in the high-resolution graphics mode, it requires enough memory to address 128,000 (640 × 200) points (i.e., 16,000 bytes). When the adaptor is in text mode, that memory is made available in the form of additional pages of text. Four pages of 80 × 25 characters, and eight pages of 40 × 25 characters, are available. A call to the video I/O routines will set the "active" page (the page that the controller currently displays). IBM's extended graphics adaptor allows for 350 lines of vertical resolution, a choice of 16 colors, downloadable fonts, and other good features. It is compatible with the function calls that drive the color/graphics adaptor.

Note that extended is a popular word and is applied to several completely different parts of MS-DOS.

EXTENDED SCREEN CONTROL

The authors of version 1 MS-DOS must have experienced pangs of guilt every now and again, until they were able to release version 2. Color and graphics features in MS-DOS computers existed, but no software had been created in MS-DOS to exercise them. Version 2 relieved a little of that guilt by adopting the ANSI standards for character output to screens that support cursor control. Microsoft calls it extended screen control. Let's examine it by first looking at what extended screen control gives us. Then we will see what we have to do to make it work for us.

The Benefits of Extended Screen Control

Extended screen control gives us the following capabilities:

- The ability to vary character attributes
- Extensive cursor control

- Partial and whole screen erasure
- Mode setting capabilities

Roughly, these capabilities let us write block-character game and business graphics in living color. We still don't have graphics capabilities, but standards for graphics packages have not yet been solidly established. Besides that impediment, flashy graphics software for 8086/8088-based computers might not benefit from (i.e., it might be slowed down by) having MS-DOS between it and the screen.

Working with Extended Screen Control

The extended-screen-control features can be exercised by (1) including the ANSI.SYS device driver in the MS-DOS configuration, and then (2) sending unique control sequences through the display output functions within interrupt 21H. The control sequences move the cursor, erase pieces of the screen, set foreground and background colors, and select text and graphics modes. We will briefly go over how to include the ANSI.SYS device driver before we look at the control sequences and how to use them.

Configuring MS-DOS to Include ANSI.SYS

ANSI.SYS is the device driver that includes the extended screen control software. The device driver needs to be part of MS-DOS for the extended screen control features to be effective.

Adding a device driver (once it is written and debugged) is easy in MS-DOS. To add this one, the line

```
DEVICE = ANSI.SYS
```

must be included in the CONFIG.SYS file. Refer to Chapter 4 to see how to add a line to the CONFIG.SYS file. Don't forget to reboot the system once the configuration file has been altered. Resident MS-DOS increases by about 1.5K bytes when this device driver is added.

Control Sequences

A control sequence is a unique string of characters that the display functions respond to. The action depends on the content of the string. Functions 1, 2, 6, 7, 9, 0AH, and 40H within interrupt 21H will respond correctly to an extended screen control sequence.

The first two characters of a control sequence are always Esc [. We will now review the various control-sequence options.

Cursor Control The cursor-control sequences are identified by three-character mnemonics. The sequences themselves are highly unmnemonic.

The CUU, CUD, CUF, and CUB sequences move the cursor from its current position one or more steps in the four cursor directions: up, down, forward (to the right), and backward (to the left). The sequences are Esc [# A, Esc [# B, Esc [# C, and Esc [# D. (Spaces are included for readability; they do not belong in the control sequences.) The pound sign (#) represents an optional decimal number of steps (default is one) specified in ASCII characters. For example, the sequence to move the cursor right two columns is

```
Esc[2C
```

The sequence to move it left one column is

```
Esc[D
```

Note that Esc stands for the single-byte ASCII code for escape, which is 1BH. Also note that there are no spaces in the examples.

CUP and HVP are two control sequences that move the cursor to Cartesian coordinates. They both have the same action: the cursor moves to the line and column specified. The sequences are Esc [# ; # H and Esc [# ; # f. The first # stands for row (line number), the second for column. For example:

```
Esc[13;40f
```

moves the cursor to about the middle of the (80-column) screen. (Why there are two different sequences for the same action is a mystery; the H sequence is not mentioned in version 3 manuals but is still available.)

The current position of the cursor can be found by issuing the DSR (device status report) sequence, which is Esc [6 n. By the way, the case of the command letter in all these control sequences is important; if the last character in this sequence is N, no cursor position is reported.

Once the DSR sequence is issued, the cursor position is reported in the form Esc [# ; # R. You can read it into a buffer using function A, buffered keyboard input. Alternatively, you can read the CPR (cursor position report) sequence by looping on function 1, keyboard input. The control sequence is conveniently terminated by 0DH, the carriage return character.

As an example, let's say you issued the DSR sequence after our cursor positioning command above. Then you used function A to read the CPR into a buffer. If you then used DEBUG to look at the buffer at location 930:200, you would see this:

```
0930:200 1B 5B 31 33 3B 34 30 52 0D . . . .[13;40R. . . . .
```

Saving and Restoring the Current Cursor Position SCP and RCP save and restore the current cursor position. Those sequences are Esc [s and Esc [u.

These sequences can be used as part of a group of sequences that saves the current cursor position, sends information to another part of the screen, and returns the cursor to its former position. For example, the save and restore sequences would be part of an operation that displays the time and date at the top right corner of the screen.

Erasure Sequences Two erasure sequences are supported. ED erases the whole screen; EL erases to the end of the line. The sequences are Esc [2 J and Esc [K.

Mode-Setting Sequences The same mode-setting capabilities are available via control sequences as can be achieved using your computer's ROM BIOS mode-setting function (function 0 within interrupt 10H [PCCs]).

The mode-setting sequence is Esc [= # h, where # is an ASCII digit, usually from 0 to 7. The correspondence between digits and modes appears in Figure 5.4. Mode 7 does not switch to monochrome. It controls whether characters typed (or output to the display by a program) after the right-hand edge of the screen is reached are moved to the next line or thrown away. If they are thrown away, any additional characters replace the last character in the line, until you stop typing (or the input buffer fills) or until the program stops sending out characters to the display.

Finally, we have reached the control sequences that make this all worthwhile—attribute byte control.

Set Graphics Rendition The SGR (set graphics rendition) sequences set an attribute byte, which is applied to all characters output thereafter. Another SGR sequence is required to change the attribute byte again. This control makes it easy to output everything in reverse video, or perhaps with high-intensity blinking if you dislike whoever is going to be reading the display.

The SGR sequences permit attribute byte selection for foreground and background colors, and, depending on your computer, reverse video, blinking, high intensity, underlined, and invisible characters. The SGR sequence is Esc [# ; # ;. . . .; m. For example, to make all subsequent characters appear in reverse video, output the SGR sequence

```
Esc[7m
```

To make all subsequent characters appear in reverse video and blinking, use the sequence

```
Esc[7;5m
```

Extended screen control does not provide all that we need. It does not support graphics, nor does it support some of the extensive video I/O

capabilities offered by the ROM BIOS function call, interrupt 10H. Let's look at those ROM BIOS capabilities as they appear in IBM PC compatibles.

VIDEO I/O USING THE ROM BIOS FUNCTION CALLS

With ROM BIOS you have almost complete control of video I/O. You can only go further by directly programming the CRT controller chips.

The ROM BIOS function calls allow you to specify (and read) the mode in which the adaptor board is to operate. This gives you the choice of operating in either the text or graphics modes. In addition, with ROM BIOS you can control the appearance of the cursor, set and read the cursor position, read and write characters with their attributes, and read and write dots (and their colors) in the graphics modes.

The fancier things that ROM BIOS allows you to do include scrolling parts (or the whole) of the screen and interacting with the light pen. The ROM BIOS functions are not all-powerful; they won't draw lines or shade areas for you in the graphics mode.

Extensions to ROM BIOS are made available when you install IBM's extended graphics adaptor. The adaptor provides additional functions under interrupt 10H that support the additional features made available by the adaptor. Only the color/graphics adaptor BIOS calls are described here.

Let's look at how to get things done using interrupt 10H.

The Format of ROM BIOS Video I/O Function Calls

ROM BIOS video capabilities are split into sixteen different functions. The choice of function is dictated by the value of the AH register on entry to interrupt 10H. To invoke a video I/O function, set the arguments appropriate to the specific function call, set the AH register to the number of the function required, and execute interrupt 10H.

Let's begin by looking at the mode-setting capability within ROM BIOS interrupt 10H. Remember, extended screen control is another way of setting the mode.

Setting the Video I/O Mode

ROM BIOS identifies seven different modes, numbered from 0 to 6. To set the mode, set AH to zero, set AL to the mode required, and execute interrupt 10H. The modes are shown in Figure 5.4.

A value of 7 is used within ROM BIOS to indicate the 80 × 25 monochrome display. The mode-reading function will return a 7 if the monochrome display is in use. The mode-setting function in ROM BIOS will

accept 7 or greater as an argument in AL to instruct it to switch back to the monochrome display from the color/graphics display.

There may be a time when you wish to know the current display mode—perhaps so that you can restore it later. Function 0FH is provided for this purpose.

Reading the Current Mode

To read the current display mode, set AH equal to 0FH (15) and execute interrupt 10H. On return, AL will contain the current mode, and AH will contain the number of columns of characters on the screen. You can infer this information from the current mode value, but you are saved the work of doing that. BH contains the number of the currently active display page if the text mode is being used.

Now that we have learned how to set and read the mode, let's look at the next most basic function: character output. You have a choice of two main functions for this purpose: (1) output of character and attribute byte and (2) output of character only. We will look at the simpler of these two functions—output of character only—after a brief introduction to the "active" page and cursor functions.

The Active Page

The active page is the page currently being displayed. We have seen that in text modes the color/graphics adaptor board contains room for either four or eight pages; only one page is active (being displayed) at any one time.

Your program can be building other pages while the user is viewing the page that is currently on display. Consequently, each time you write characters, the page must be indicated. The page is one argument to the output

AL	Mode
0	Text, 40 × 25, black and white
1	Text, 40 × 25, color
2	Text, 80 × 25, black and white
3	Text, 80 × 25, color
4	Graphics, medium resolution, color
5	Graphics, medium resolution, black and white
6	Graphics, higher resolution, black and white

Figure 5.4: *Display Modes Recognized by ROM BIOS*

character function calls. When you want another page to be displayed, the function to set the active page can be invoked.

Setting the Active Page

The function to select the active page is function AH = 5, within interrupt 10H. The page number is passed in AL; its value is 0 to 7 for 40 × 25 modes (modes 0 and 1), and 0 to 3 for 80 × 25 modes (modes 2 and 3).

To set the active page, put the page number in AL, set AH equal to 5, and execute interrupt 10H. The newly selected page is instantly displayed.

The Current Cursor Positions

The ROM BIOS functions write characters to a page's current cursor position. Eight cursor positions are maintained—one for each of the eight possible pages. Once you have set the cursor position (function AH = 2), it is maintained by the routines that write characters to the page.

Before we move on to discuss character output, we will look at both setting and reading the cursor position and altering the appearance of the cursor. Remember, MS-DOS version 2 or later can be configured to accept cursor control commands; you may not need these ROM BIOS alternatives.

Setting the Cursor Position

Function AH = 2 sets the cursor position. The arguments to the function are the row and column (numbered from 0,0 in the top left corner) and the page on which the cursor is to be set. You can set the cursor position while operating in either the text or graphics mode. The function always understands the cursor-position arguments to be text coordinates (varying from 0 to 25 and 0 to 80), not graphics coordinates. In the graphics mode the page argument must be zero.

To set the cursor, put the page number in BH, the row in DH, and the column in DL; then set AH equal to 2 and execute interrupt 10H.

Reading the Current Cursor Position

This function (AH = 3) reads the current cursor position in the page specified. It also tells you what the cursor looks like. To read the cursor position, set BH to the page number you want to look into, set AH equal to 3, and execute interrupt 10H.

On return, DH and DL are the row and column positions of the cursor. If you have previously changed the appearance of the cursor, CH and CL will show you the start and end line for the cursor. You may be wondering what the "start and end line for the cursor" means. Please read on.

Changing the Appearance of the Cursor

The cursor's appearance can be made quite different from the thick underline character that you usually see. Function AH = 1 allows you to control the upper and lower limits of the blob that blinks. You can move it upward, enlarge it, split it into two parts, or shrink it so that it looks like the underline character.

To understand what "start and end line" means, we need to explain what horizontal scan lines are. If you already know, you can skip the next paragraph.

Images on monitors are made by zipping a narrow electron beam across the screen—turning it on when you want the phosphor to glow and leaving it off for dark spots. The beam is thin, so to cover the whole screen it has to trace a path across the screen many times. The beam starts at the top of the screen, moves across (and back), then moves down a distance slightly greater than its thickness, and moves across the screen again. The path traced by the beam across the screen is called the *horizontal scan line*.

Several (the number depends on the resolution of the display adaptor; we will use 14 here) horizontal scan lines are devoted to each line of characters. They are numbered from 0 through 13, starting at the top. The normal monochrome cursor, which is two lines thick, can be produced by starting it at line 12 and ending it at the last line, line 13.

A cursor designed to occupy a whole rectangle can be produced by instructing ROM BIOS to start drawing it at line 0 and stop drawing at the last line. A split cursor, where blinking lines appear at both top and bottom of the character rectangle, can be obtained by setting CH, the start line, to 12, and CL, the end line, to 0. Note that these values apply only to a display driven by an adaptor that supports 14 horizontal scan lines per line of characters. On screens driven by the color/graphics adaptor, there are 8 lines per character rectangle [PCCs], numbered from 0 to 7 from the top down.

To change the appearance of the cursor, set CH to the start line for the cursor, CL to the end line, and AH equal to 1, and then execute interrupt 10H. The cursor shape is set for all pages at once.

Now let's look at the functions that write characters to a page.

Character I/O Using ROM BIOS

ROM BIOS provides three different functions for sending characters to a page, and one function that reads the character at the current cursor position. We will examine the two major output functions first: character output without attribute and character output with attribute. A note on the limitations of these character output functions when the color/graphics adaptor is operating in either of the two graphics modes (5 or 6) is included.

Character Output, No Attribute

This function is the simpler of the two major character output functions, but only in that it does not require the character's attribute byte to be passed as one of the arguments. With this output function, any attribute bytes that exist in the adaptor's memory will be used "as is." Like the other character-writing function call, this one will output the same character more than once if required.

We have now been through enough background information to understand the arguments needed by the character output functions. To output one or more of the same characters, starting at the current cursor position, use the function addressed by AH = 0AH. Set BH to the display page you wish to write to, set CX to the count of characters to write, put the character to write in AL, set AH to 0AH, and execute interrupt 10H.

Character Output, with Attribute

This output function uses one more argument than the simpler function, and that argument contains the attribute byte. As before, the character, with its attribute byte, is output to the page of choice at the current cursor position. Several copies of the character may be output.

Set AL to the character to write, set CX to the number of characters, set BH to the display page, set BL to the attribute byte, set AH equal to 9, and execute interrupt 10H.

Character I/O in the Graphics Mode

If you wish to output characters to the screen while in the graphics mode, ROM BIOS will do extensive work for you. In the graphics mode the character writing functions use a built-in table of dot patterns and output each character dot-by-dot. These functions, however, contain the information to do this only for the first 128 ASCII characters. You can point interrupt 1FH (locations 7C through 7F) to your own table of dot positions for the ASCII characters 128 and up.

If you wish to output multiple characters in the graphics mode, the performance of the two major character-writing functions is not guaranteed beyond the end of the line on which the output begins.

Read the Character at the Current Cursor Position

It may sometimes be useful to read the character (and attribute) in the display buffer. Function AH = 8 does this for you.

The only argument for this interrupt 10H function is the display page. To read the character and its attribute byte, set BH to the page of interest, set AH = 8, and execute interrupt 10H. On return, the character at the current cursor position for that page will be in AL; the attribute byte will be in BL.

The Third Character Output Routine

We have talked about two "major" character output routines. Well, here is another one, a minor one. It is a comparatively simple character-output routine; in fact, it is the equivalent of teletype output. The character is in AL, the page is in BH, and if we are operating in the graphics mode, the foreground color is in BL. You will need to set AH equal to 0EH, then execute interrupt 10H. A single character will be output to the current cursor position.

There are yet more capabilities built into the ROM BIOS video I/O function. Scrolling is one of them, and it's fun to play with. Let's look at the scrolling facilities.

Scrolling with ROM BIOS

The tools exist for scrolling whole pages or parts of pages either up or down. When you call upon function AH = 6 (scroll up) or AH = 7 (scroll down), the window in the active page (the window is defined by the row and column positions of the upper left and lower right corners) scrolls up or down by the number of lines specified. The lines (or partial lines) scrolled "away from" are blanked and have the specified attribute byte added to them.

To scroll, set AL to the number of lines (zero means scroll the whole window, i.e., blank it), set CH and CL to the row and column position of the upper left corner, set DH and DL to the lower right coordinates, and set BH to the attribute byte to put into the blanked lines. When all the arguments are ready, set AH equal to 6 to scroll up, or 7 to scroll down, and execute interrupt 10H. That piece of the screen should then scroll as directed.

Now that we have covered character output, let's look at the graphics capabilities in ROM BIOS.

Graphics Output in ROM BIOS

Three functions are available for graphics output—write dot, read dot, and color control. The write and read functions are almost self-explanatory; the color-control function requires a little attention, so we will look at it first.

To set the color palette, function AH = 0BH, you need to understand the color control available via the color/graphics card. Color control often applies only to medium-resolution graphics. Basically, you have a choice of sixteen colors for the background and two sets of three colors for the foreground.

Setting the Color Palette

Let's learn how to establish color choices using function 0BH. You need to specify colors for two different parts of the screen: the background and

the foreground. Register BH is used to specify whether it is background or foreground you wish to set a color (or colors) for. Once you have established that, you need to indicate the color choices. This is done via register BL.

To choose one of sixteen colors for the background, set BH to 0 and BL to the color of your choice. To choose one of the two color palettes available for the foreground image, set BL to 0 to get the green, red, and brown (actually sort of yellow) group, or set BL to 1 to get the cyan, magenta, and white group. Once you have the BH and BL arguments ready, set AH equal to OB and execute interrupt 10H.

Write a Dot in Graphics Mode

All you have to do to write a dot is specify the location and, if operating in medium-resolution mode (320 × 200), the color of the dot.

The location is specified in terms of row and column. The row coordinate varies from 0 to 199 in both medium- and high-resolution modes. The column coordinate varies from 0 to 319 in medium resolution, and from 0 to 639 in high resolution. Location 0,0 is the top left corner of the screen; location 319,199 or 639,199 is the lower right corner.

The color of the dot can be any one of four: the background color or one of the three foreground colors. If you don't write a dot, it will be the background color.

To write a dot, set DX to the row, set CX to the column, set AL to the color (0, 1, 2, or 3), set AH equal to 0C, and execute interrupt 10H.

Read a Dot in Graphics Mode

Reading a dot gets you the color of the dot at the position you specify. The color is returned in AL. To read a dot, set DX to the row, CX to the column and AH equal to 0D, and execute interrupt 10H.

There is one last capability of the ROM BIOS video functions to be revealed: the light-pen function.

Reading the Light-Pen Position

If you want to know where the light pen is currently positioned on the screen, set AH equal to 4 and execute interrupt 10H. On return, if AH = 0, the light pen switch is not triggered, so no location is made available.

If AH = 1, you have the light-pen position expressed in two forms. The row and column of the character position are in DH and DL. DH will vary from 0 to 25, and DL from 0 to 80. A more refined position is returned in the CH and BX registers. CH contains the horizontal scan line (alias raster scan row), varying from 0 to 199; BX contains the pixel column, varying from 0 to 319 or 639.

LOCATIONS OF THE CHARACTERS AND THEIR ATTRIBUTE BYTES

This section is for those of you who want to directly access the character buffers in the monochrome and color/graphics adaptors on an IBM PC compatible. Short of programming the CRT controller directly, this approach may be the quickest way to modify an image on the screen.

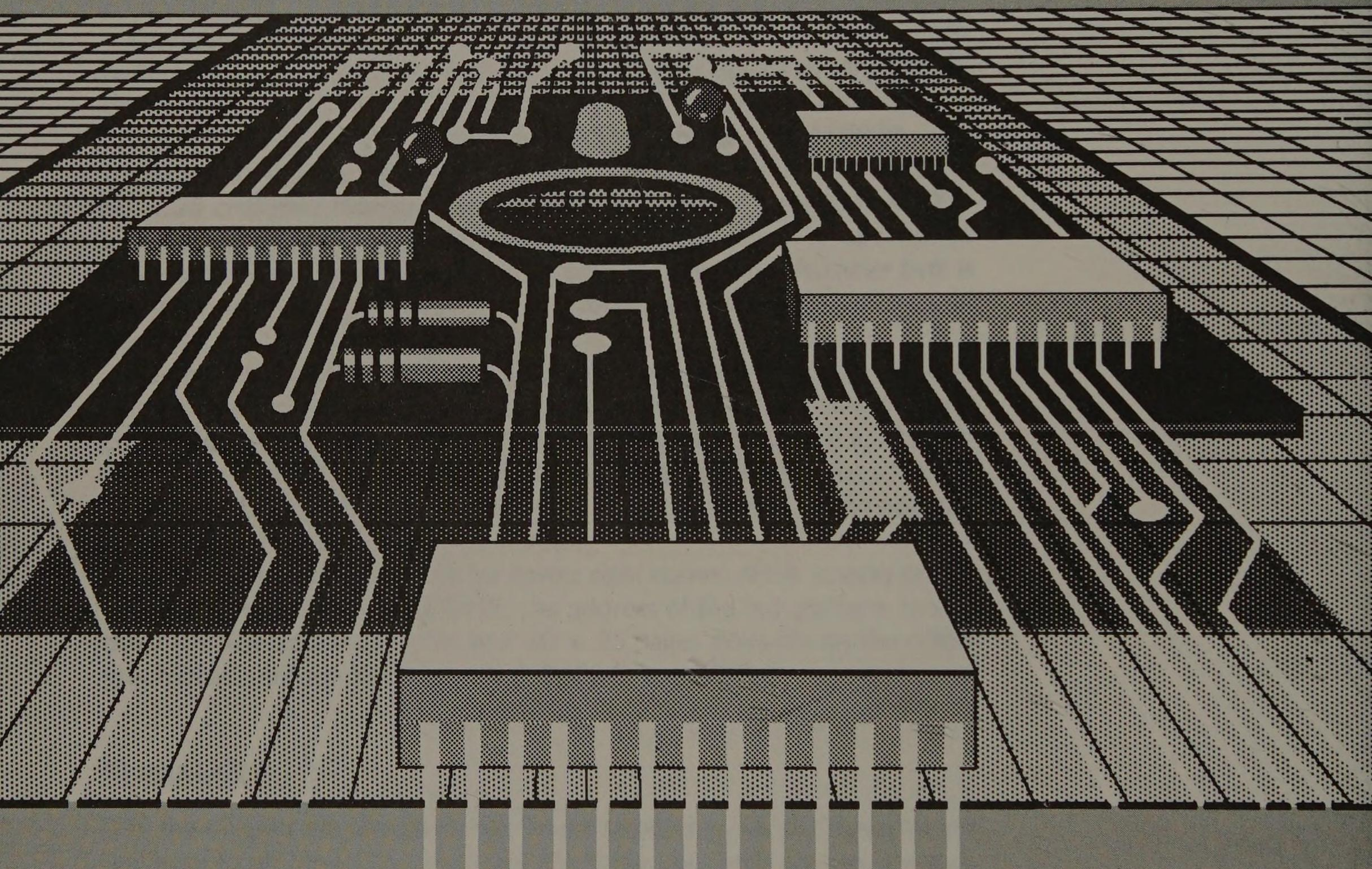
The attribute byte follows the character byte in the display buffer. An 80 × 25 character display will show 2000 characters. It requires a 4000-byte buffer, however, because each character on the screen is represented in the buffer by a character byte *and* an attribute byte. The character byte is found on an even address and the attribute byte on the following odd address.

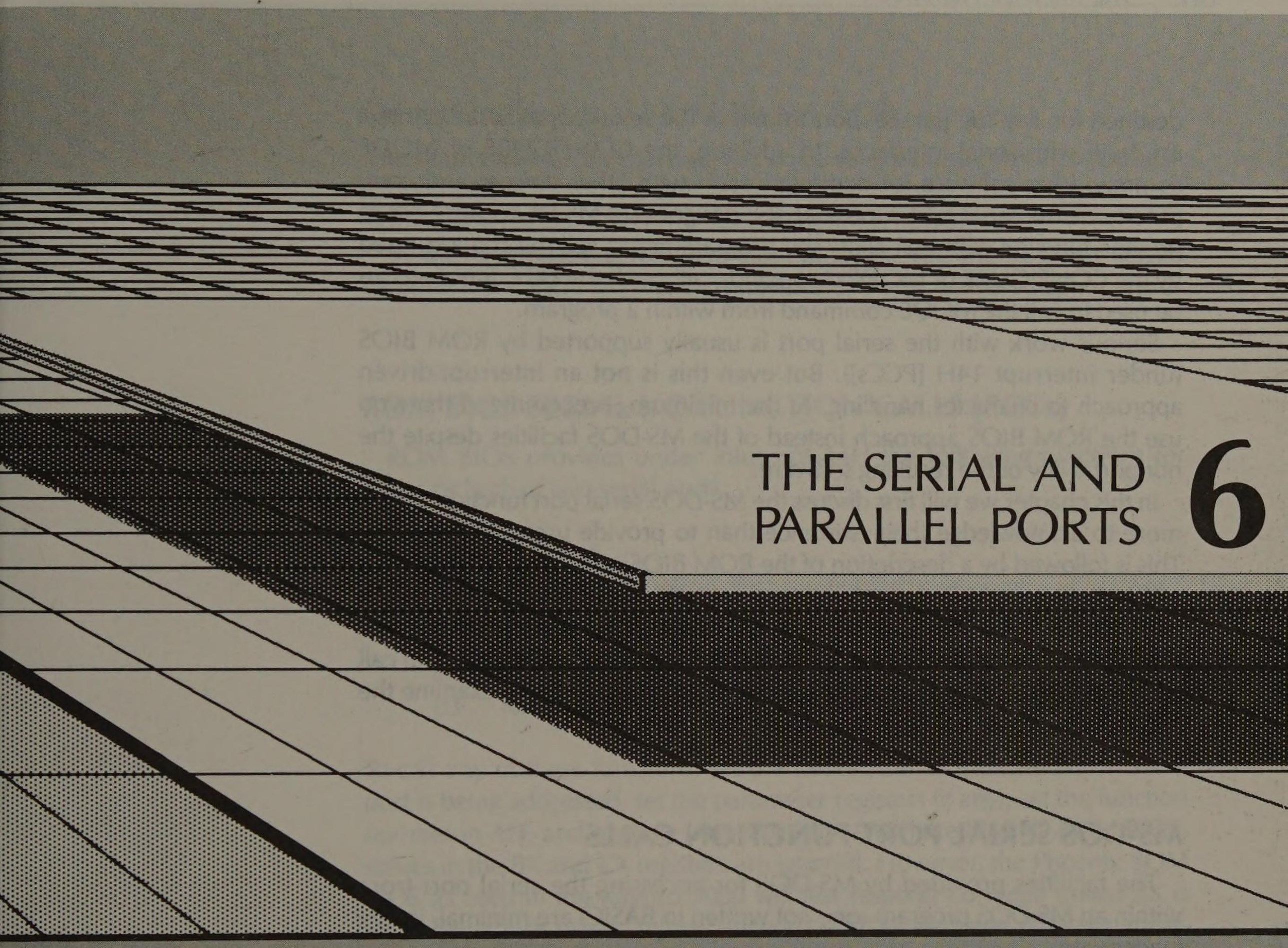
The buffer space for both monochrome and color/graphics displays exists on the respective adaptor. The address of the beginning of the monochrome display buffer is B0000H, the beginning of the color/graphics display buffer is at B8000H. Since the attribute byte follows the character, the address of the first attribute byte is B0001H on the monochrome adaptor, or B8001H on the color/graphics adaptor.

The address of the last character (lower right corner of the screen) on the monochrome adaptor is B000:F9E; the address of the last attribute byte is B000:F9F. The addresses of the four 80 × 25 pages available on the color/graphics adaptor are B800:0, B900:0, BA00:0, and BB00;0.

SUMMARY

In this chapter we have seen the simple screen-access functions that are universal to all MS-DOS computers. We have also seen how to program output for the displays used by some of the IBM PC compatibles. We also examined character attribute bytes and their effect on the character in both monochrome and color/graphics modes. We reviewed extended screen control and then explored the ROM BIOS video I/O capabilities in some detail. These capabilities include almost all the functions you may want when programming for output to a display.





THE SERIAL AND PARALLEL PORTS 6

INTRODUCTION

This brief chapter discusses the tools the programmer has for conducting input and output via the serial* and parallel ports that can be incorporated into your MS-DOS computer. It is brief because MS-DOS is not strong on data transfer via these two ports at the programmer's level.

If you are familiar with the CONFIGURE (or MODE [PCCs]) command, you might expect better tools to be available to programmers. The CONFIGURE or MODE command allows the user to redirect output originally

---

* IBM calls the serial port an Asynchronous Communications Adaptor.

destined for, say, the parallel port to one of the serial ports. Some printers are built with serial interfaces. In addition, the CONFIGURE or MODE command has software for communication with other devices and computers, via the serial port. However, the designers of MS-DOS chose not to provide function calls that allow the programmer to call the routines used by the CONFIGURE or MODE commands, although the EXEC function can be used to call the MODE command from within a program.

Serious work with the serial port is usually supported by ROM BIOS (under interrupt 14H [PCCs]). But even this is not an interrupt-driven approach to character handling. At the minimum, I recommend that you use the ROM BIOS approach instead of the MS-DOS facilities despite the nonportability of the resulting software.

In this chapter we will first discuss the MS-DOS serial port function calls—more to acknowledge their existence than to provide useful information. This is followed by a description of the ROM BIOS serial port function calls within interrupt 14H, thus giving you something you can actually work with. We will then move on to a note on the hazards of programming for RS-232 serial communications. We will also describe the single function call within MS-DOS for output to the parallel port. Finally, we will examine the ROM BIOS printer I/O function calls under interrupt 17H.

MS-DOS SERIAL PORT FUNCTION CALLS

The facilities provided by MS-DOS for accessing the serial port from within an MS-DOS program (one not written in BASIC) are minimal, in fact inadequate.

MS-DOS provides two function calls, numbered 3 and 4, within interrupt 21H for single-character, unbuffered, noninterrupt-driven input and output. Let's examine these two serial-port function calls.

Input from the Serial Port with MS-DOS

To get a character from the serial port, set AH equal to 3 and execute interrupt 21H. The character is returned in AL.

With this function call, the input is unbuffered, so your software had better be there waiting for a character any time one is likely to arrive, or input characters will be missed. This means that, among other complications inherent in this approach to serial-port management, the time your program takes to process a character, do whatever else it has to do, and then call function 3 in interrupt 21H (to go wait for the next character) must be less than the interval between characters arriving at the port.

Output to the Serial Port with MS-DOS

To output to the serial port, set AH equal to 4, put the character you wish to output into DL, and execute interrupt 21H. The character in DL is output to the first serial port. No error conditions are returned.

As with the matching MS-DOS input function, you will be better served by using ROM BIOS facilities for serial-port I/O.

ROM BIOS SERIAL I/O FUNCTION CALLS [PCCs]

ROM BIOS provides under interrupt 14H the following functions for either or both of two serial ports:

AH	Function
00	Initialize baud rate, parity, word length, and number of stop bits
01	Output a character
02	Receive a character
03	Read port status

To call any of these functions, set the DX register to indicate which serial port is being addressed, set the parameter registers (if any), set the function number in AH, and execute interrupt 14H. (With the IBM PC ROM BIOS, values in the BX and CX registers are ignored. However, the Phoenix ROM BIOS as used in the Kaypro 2000 will not respond correctly unless BX is zero. Beware of other "minor" incompatibilities between different implementations of ROM BIOS.) Let's look at the four functions, starting with the one that initializes a serial port.

Serial Port Initialization

This function specifies the baud rate, parity, word length, and number of stop bits that the serial port's hardware and software have to handle. These four parameters are all defined within the AL register: the baud rate is defined in the three high-order bits (bits 7–5), the parity in the next two bits (bits 4–3), the number of stop bits is defined by bit 2, and the word length is defined in the last two bits (bits 1–0). Figure 6.1 illustrates this arrangement. Figures 6.2 through 6.4 show the meanings of the bit values for the baud-rate bits, the parity bits, and the word-length bits. For the stop-bits parameter, if one stop bit is to be used, bit 2 is zero; if two stop bits are required, bit 2 is set.

(Note: If you want to communicate at 19,200 baud, you will need a serial port that can be programmed to run that fast (several serial boards support 19,200 baud), and you will need to address the 8250 controller chip directly. *The IBM Technical Reference* that describes the Asynchronous Communications Adaptor supplies a large amount of not very decipherable information about this device. As usual, the information is all there; it is just that no path has been provided through it all.)

Send a Character

Function 1 within interrupt 14H sends out the character in AL. On return, if there was an error, bit 7 of AH is set and the rest of the bits in AH contain the serial port status, the same status as is returned by the read status function.

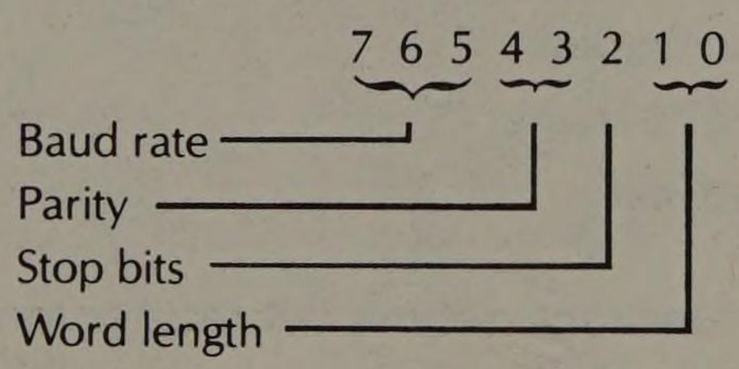


Figure 6.1: *Serial Port Initialization Parameters in Register AL*

Bits 7–5	Baud Rate
0 0 0	110
0 0 1	150
0 1 0	300
0 1 1	600
1 0 0	1200
1 0 1	2400
1 1 0	4800
1 1 1	9600

Figure 6.2: *Baud Rate Bit Definitions for Serial I/O*

Receive a Character

Function 2 within interrupt 14H waits for a character from the serial port. If AH is nonzero on return, an error has occurred, and bits 7, 4, 3, 2, and 1 of AH are set, as described in the read status function. If AH is zero, the character received is in AL.

Read Serial Port Status

Function 3 within interrupt 14H reads the serial port status. Figure 6.5 shows the conditions returned.

Setting Modes from within a Program

The EXEC function (number 4B of interrupt 21H) permits MS-DOS commands to be called from within an applications program. It takes a little setup work, but the facility is there to do within your software anything that you can do with, say, the MODE command from the keyboard. The minor disadvantage of this approach is that the EXEC function brings in another copy of the command processor, which is about 4K of additional space you have to find room for.

Bits 4, 3	Parity
0 0	None
0 1	Odd
1 0	None
1 1	Even

Figure 6.3: *Parity Bit Definitions for Serial I/O*

Bits 1, 0	Word Length
1 0	7 bits
1 1	8 bits

Figure 6.4: *Word Length Bit Definitions for Serial I/O*

THE HAZARDS OF PROGRAMMING FOR RS-232 COMMUNICATIONS

Routines capable of handling bidirectional communication over serial ports generally end up being much more extensive in their capabilities, and therefore requiring much more programming effort, than at first expected. The problems with programming such routines are twofold: (1) your program does not control both ends of the transfer, so it has to deal with an unusual variety of logical error conditions; (2) the integrity of the electronic path over which the signals travel is often poor, so the routines have to be able to issue, and honor, retransmission requests after checksum errors have been detected.

The best way to handle serial communication is to address the 8250 controller chip (at-3F8H for COM1 and 2F8H for COM2), directing it to interrupt on each character.

You might think of using I/O redirection to send and receive data via serial ports. This will work in a limited number of situations. It is not, however, a general solution. Serial communication is not simply a case of shoveling characters out to the serial port instead of, say, the display. The display

AH Bit	Line Control Status
7	Time out
6	Transmitter shift register empty
5	Transmitter holding register empty
4	Break detected
3	Framing error
2	Parity error
1	Overrun error
0	Data is ready

AL Bit	Modem Status
7	Received line signal detect
6	Ring indicator
5	Data set ready
4	Clear to send
3	Receive line signal detect changed since last checked
2	Trailing edge of ring detector
1	Data set ready changed since last checked
0	Clear to send changed since last checked

Figure 6.5: *Status Conditions, ROM BIOS Serial I/O Function*

does not talk back, nor does it need communications protocol characters inserted in the text to continue operations over any appreciable period of time. The applications need to be totally simple for I/O redirection to provide a communications solution.

Should you be discouraged from trying to write serial I/O routines, take heart. If you have, or can borrow, a copy of the *IBM Technical Reference,* you will find helpful examples provided by the ROM BIOS routines. For issues concerning the correct cabling of devices using the serial ports, see Chapter 16 of this book. When you write serial I/O routines, remember to use interrupt-driven, buffered I/O instead of character-by-character transmission directly from the noninterrupt levels of your program.

The excellent book *The RS-232 Solution* by Joe Campbell (SYBEX, 1984) will greatly assist your serial I/O programming efforts. It offers much more insight than can be provided here.

THE PARALLEL PRINTER INTERFACE

MS-DOS does not provide programmers with extensive software tools for handling output to the parallel port. It assumes that error conditions can be ignored, and that there is only one printer. ROM BIOS provides better functions for parallel printer control.

Let's first examine the MS-DOS function call for output of a character to that one printer. Then let's examine the ROM BIOS function calls, which are far more satisfactory for driving a parallel printer.

Output to the Printer with MS-DOS

Function number 5 in interrupt 21H is the single function call provided for sending a character to the printer. To use this function, set AH to 5, put the character to be output into DL, and execute interrupt 21H. The character in DL is sent to the first printer. No error conditions are reported.

Output to the Printer Using ROM BIOS [PCCs]

The IBM PC compatibles' ROM BIOS contains three functions under interrupt 17H that are capable of addressing three separate parallel ports, as follows:

AH	Function
00	Print the character in AL, with provision for completion checking
01	Initialize the port (one configuration only); status returned
02	Read the printer status

DX is used to specify which printer to use—0, 1, or 2. Let's look a little more closely at these three functions.

Print the Character in AL

Function 0 within interrupt 17H prints the character in AL. On return, AH contains the status bits described in the read-status function.

Initialize the Printer Port

Function 1 within interrupt 17H initializes the printer port. The printer's response depends on how it interprets the output of this function. If you have an Epson dot-matrix printer, you may have noticed it twitch when you boot MS-DOS; that is caused by this function.

Read the Printer Status

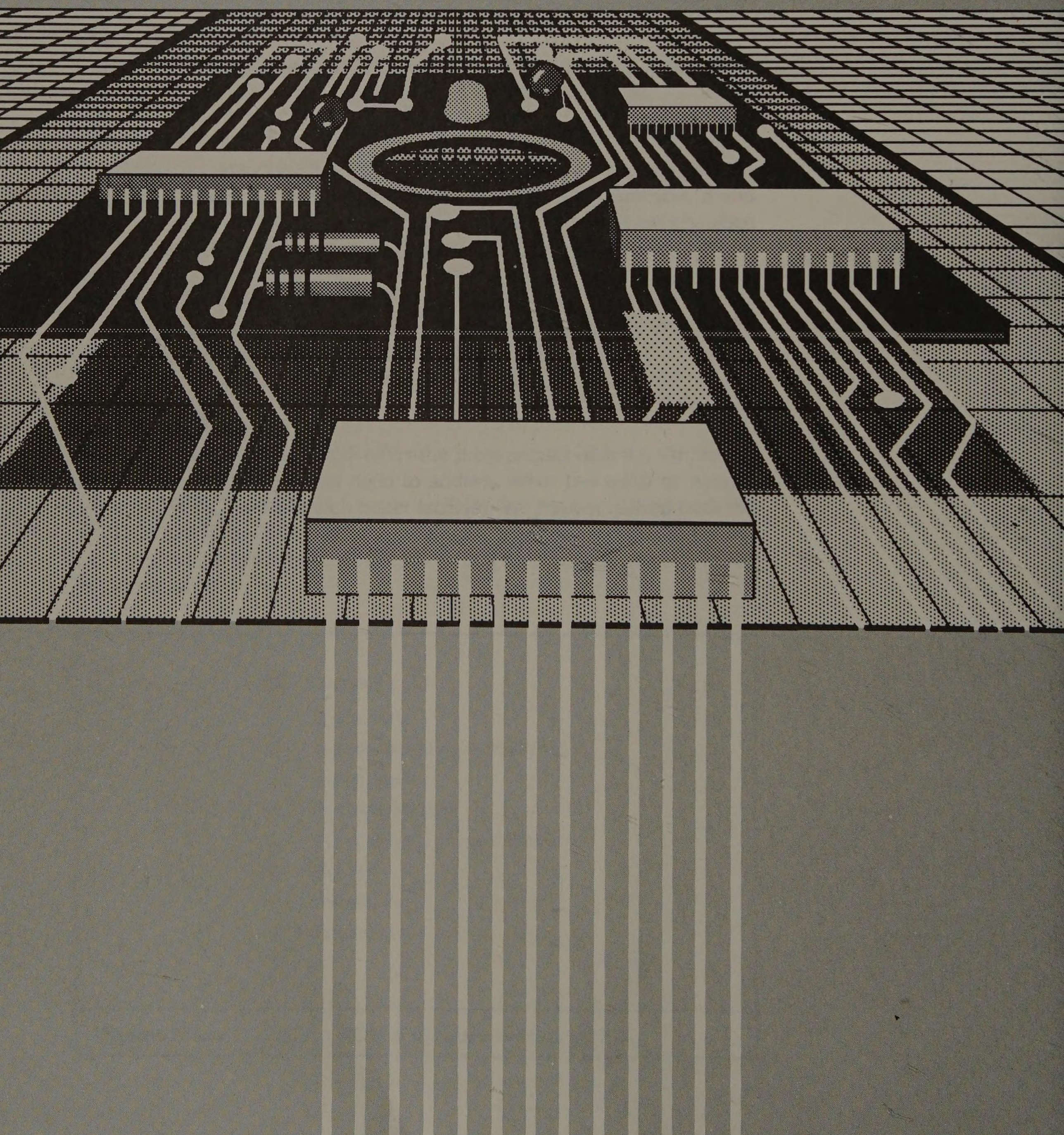
Function 2 within interrupt 17H returns the printer status in AH. The meaning of the status bits is shown in Figure 6.6.

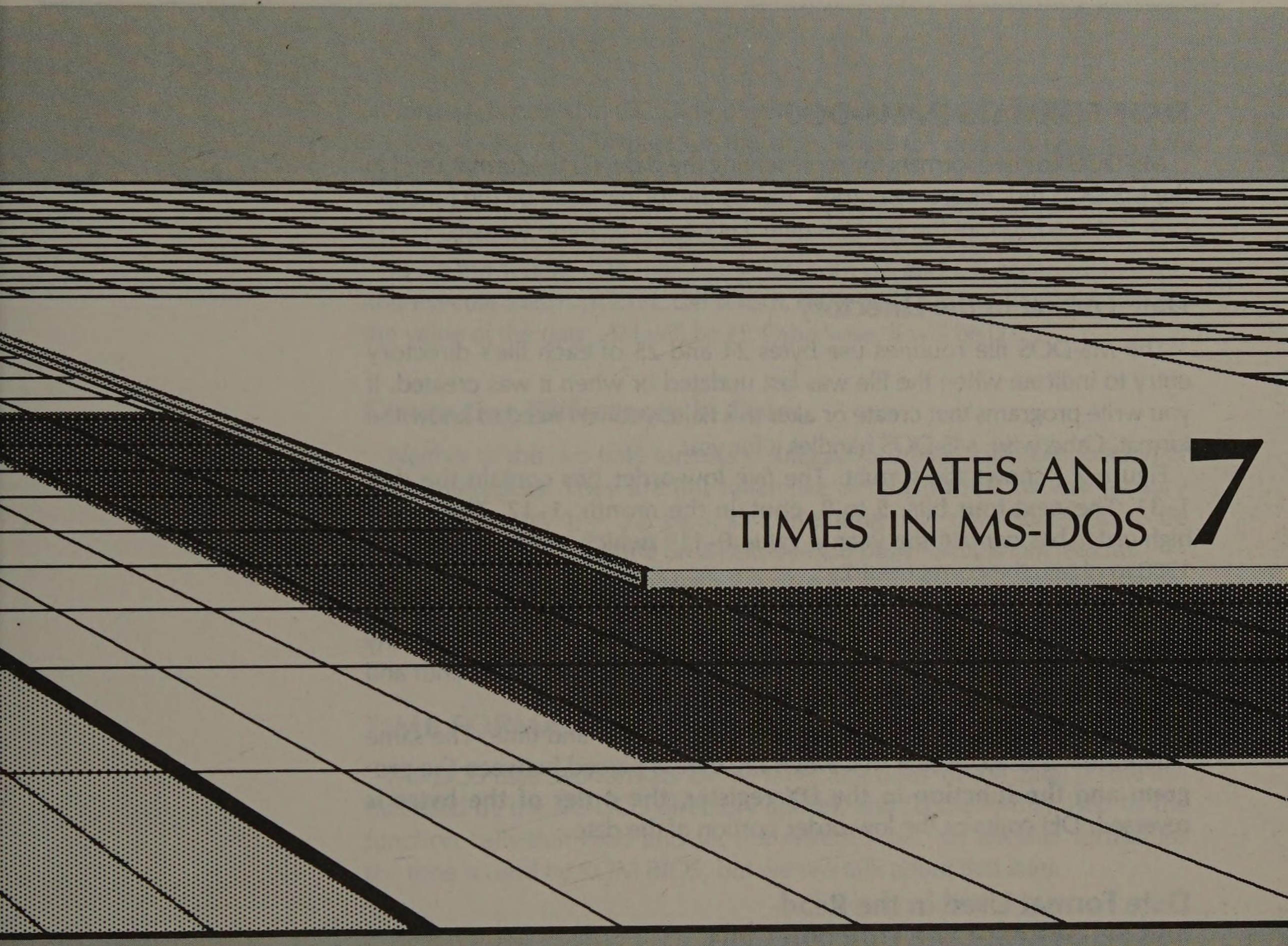
SUMMARY

We have seen how MS-DOS offers the programmer only the simplest of facilities for applications that need to address either the serial or parallel ports. ROM BIOS offers much better facilities. We have reviewed both the MS-DOS and ROM BIOS functions. We noted that programming at the I/O port level is complex for serial I/O control, and somewhat simpler for parallel printer control.

Bit	Meaning
7	Printer busy
6	Acknowledge (not normally set)
5	Out of paper
4	Printer is selected
3	I/O error (e.g., printer off line)
2	Unused
1	Unused
0	Time out

Figure 6.6: *Status Conditions, ROM BIOS Parallel Printer Control*





DATES AND TIMES IN MS-DOS 7

INTRODUCTION

This chapter presents the different formats used for representing date and time in MS-DOS. It describes the MS-DOS function calls that read and set the date and time from within a program. In addition, it discusses the ROM BIOS time-of-day function (interrupt 1A) and the interval-timer functions (interrupts 1C and 08) that are provided in the IBM PC compatibles. Notes are included on the accuracy that can be expected from the time routines. The chapter concludes with a discussion of how your software can be interrupted at every tick of the timer, and how you can have total time control, at a price.

DATE FORMATS IN MS-DOS

MS-DOS has two formats for representing the date: (1) the format used in the MS-DOS disk directory entries, and (2) the format used by the function calls that read and set the system's date. Let's examine these formats.

Date Format in the Directory

The MS-DOS file routines use bytes 24 and 25 of each file's directory entry to indicate when the file was last updated or when it was created. If you write programs that create or alter this field, you will need to know the format. Otherwise, MS-DOS handles it for you.

Figure 7.1 shows the format. The five low-order bits contain the day, 1–31. The next four bits, 5 to 8, contain the month, 1–12. The seven high-order bits contain the year, a value 0–119 (which must be added to 1980 to obtain the correct year).

If the date is not set on a system restart, i.e., power-on or system reboot (Ctrl-Alt-Del key combination [PCCs]), and if the files are subsequently altered or created, the year field will have a zero value and the month and day fields will be set to one.

Function 57H allows you to read or set a file's date and time. The same date format is used, **except that when the date is passed between the program and the function in the DX register, the order of the bytes is reversed.** DH contains the low-order portion of the date.

Date Format Used in the Read- and Set-System-Date Function Calls

Function calls 2A and 2B within interrupt 21H get and set the date. These routines use the CX and DX registers to pass the date. The year, 1980–2099

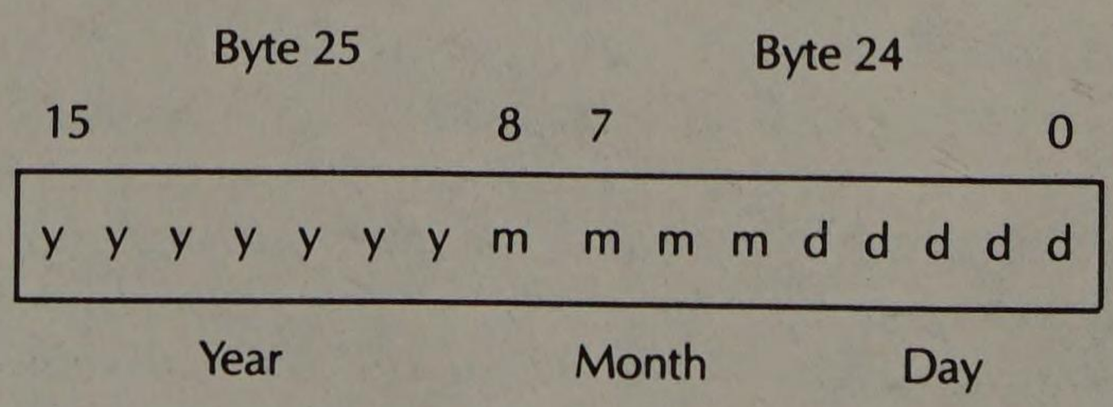


Figure 7.1: *Format of a Date in the Directory Entry*

(in binary), is stored in CX. DH contains the month, 1–12, and DL contains the day, 1–31. The formats for the date in the CX and DX registers are shown in Figure 7.2.

To read the system date, you set AH equal to 2A and execute interrupt 21H. On return, CX will contain the year; DX, the month and day.

To set the system date, use AH = 2B, with CX and DX appropriately set, and execute interrupt 21H. On return, if the MS-DOS routines did not like the value of the date, AH will be FF. Otherwise, it will be 00.

Computing Differences in Dates

Neither of the two date formats is suitable for immediate computation of date differences. They are not Julian-like in their format (where January 1 = 1, February 1 = 32, etc.), so computing date differences has to be done by your own routine on a field-by-field basis—simple but tedious.

Now that we are familiar with the date formats, let's discuss the different representations of time.

TIME FORMATS IN MS-DOS

MS-DOS uses two formats for the time, as it does for the date: (1) the format used by the MS-DOS directory entries, and (2) the format used by the function calls that read and set the system time. Yet another format for the time is used by ROM BIOS, but we will talk about that later.

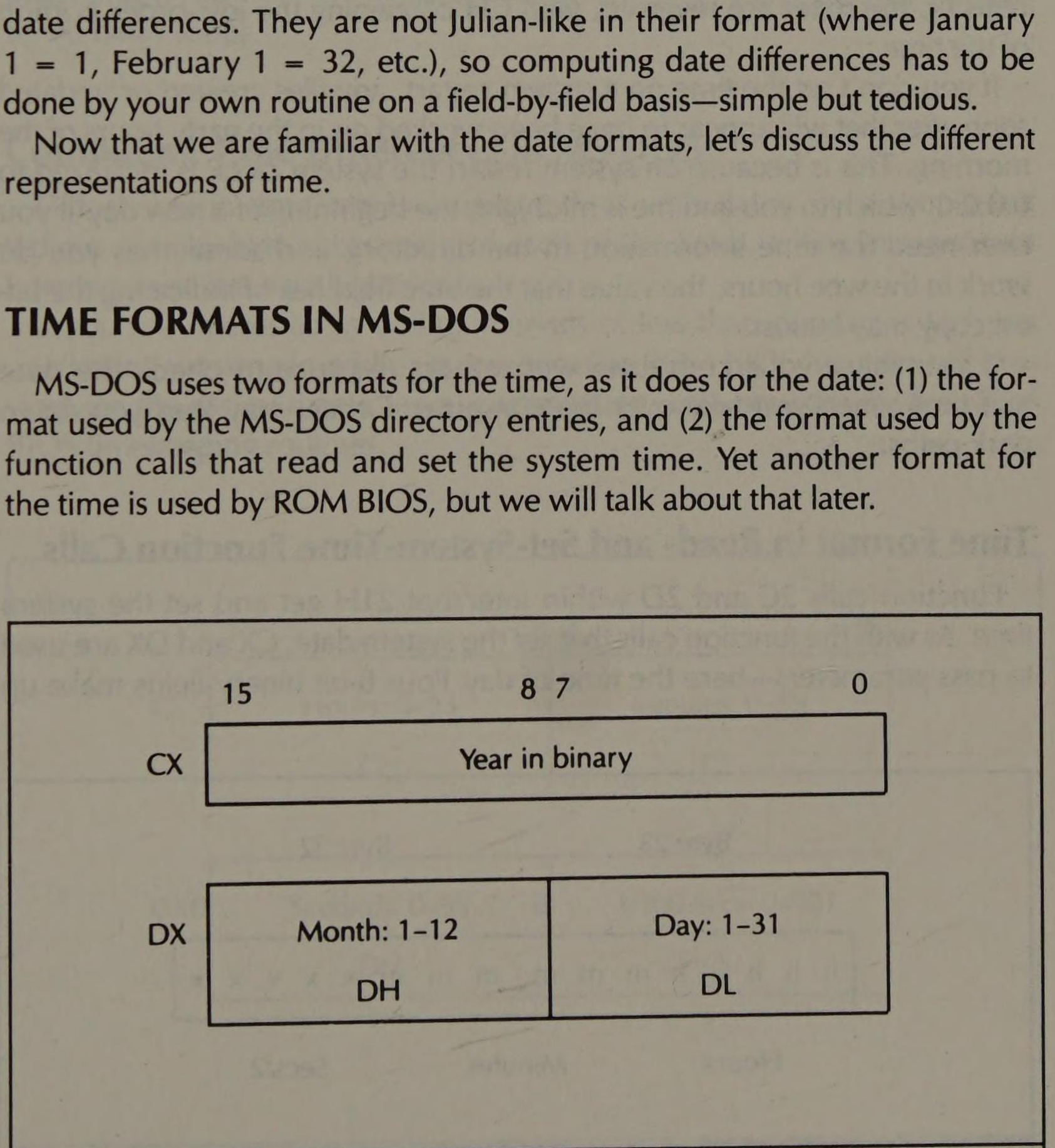


Figure 7.2: *Format for the Date Used in the Set-System-Date Function Call*

Time Format in the Directory

The MS-DOS file routines maintain bytes 22 and 23 in each directory entry to indicate the time when the file was last updated, or when it was created (if it has not been updated). The internal format of each directory entry's time field is shown in Figure 7.3. The five low-order bits, 0–4, contain the number of two-second increments. (That two-second interval is a small concession to having to fit the time of day into sixteen bits.) The next six bits, 5–10, contain the minutes, 0–59; the five high-order bits contain the hours, 0–23.

Function 57H within interrupt 21H can be used to get or set a file's time (and date). As with the date, **when the time argument is passed in the CX register, the bytes are reversed,** with CH containing the low-order portion of the time.

If you don't set the time on a system restart, any files created or updated soon after that will appear to have been worked on in the early hours of the morning. This is because on system restart the system clock is initialized to 0:0:0.0, which to you and me is midnight, the beginning of a new day. If you ever need the time information in the directory, and sometimes you do work in the wee hours, the value that the time field has of indicating the latest copy may be lost.

If you share your computer with others, be sure to check the date and time when you take over the computer. Others may lead you down dark paths.

Time Format in Read- and Set-System-Time Function Calls

Function calls 2C and 2D within interrupt 21H get and set the system time. As with the function calls that set the system date, CX and DX are used to pass parameters—here the time of day. Four 8-bit binary fields make up

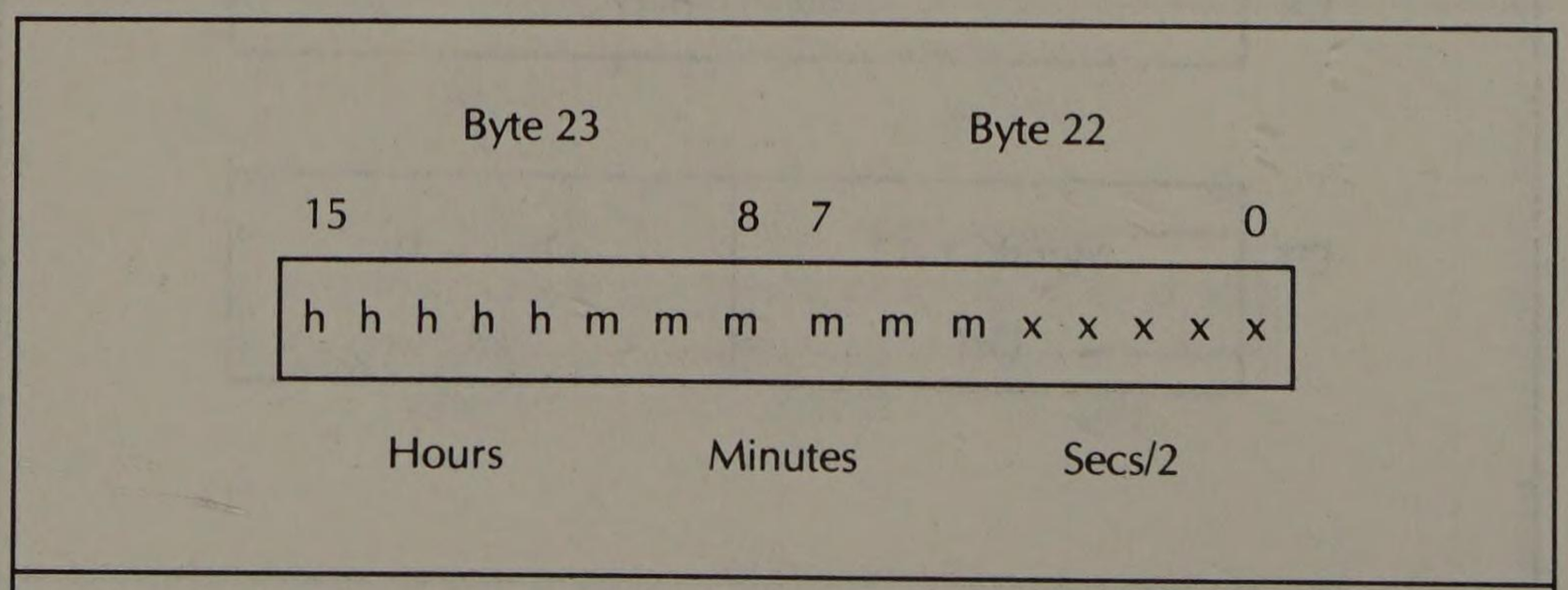


Figure 7.3: *Format for the Time in the Directory Entry*

the time. CH contains the hours, 0–23; CL contains the minutes, 0–59; DH has the seconds, 0–59; and DL contains the hundredths of seconds, 0–99. Figure 7.4 shows the formats for the CX and DX registers.

Accuracy in the MS-DOS Time Format

In the IBM PC compatibles, MS-DOS uses the ROM BIOS timekeeping routines, so it is a little presumptuous when it pretends to an accuracy of 1/100th of a second. The time as expressed by ROM BIOS will be, on average, off by about 1/36th second in either direction. Therefore, relying on it to the nearest 1/100th second is not appropriate. If you need that kind of accuracy, you will have to build your own timing routines, using carefully timed delay loops.

Computing Time Differences in MS-DOS

Differences in time can be computed, as with dates, by a simple but tedious byte-by-byte subtraction of each of the fields, taking into account borrows from the next higher byte.

If you are interested in timing intervals of less than one hour, and you don't mind working in odd units, perhaps the ROM BIOS function call provides for the easiest time interval computation. Let's now discuss the ROM BIOS timekeeping routines.

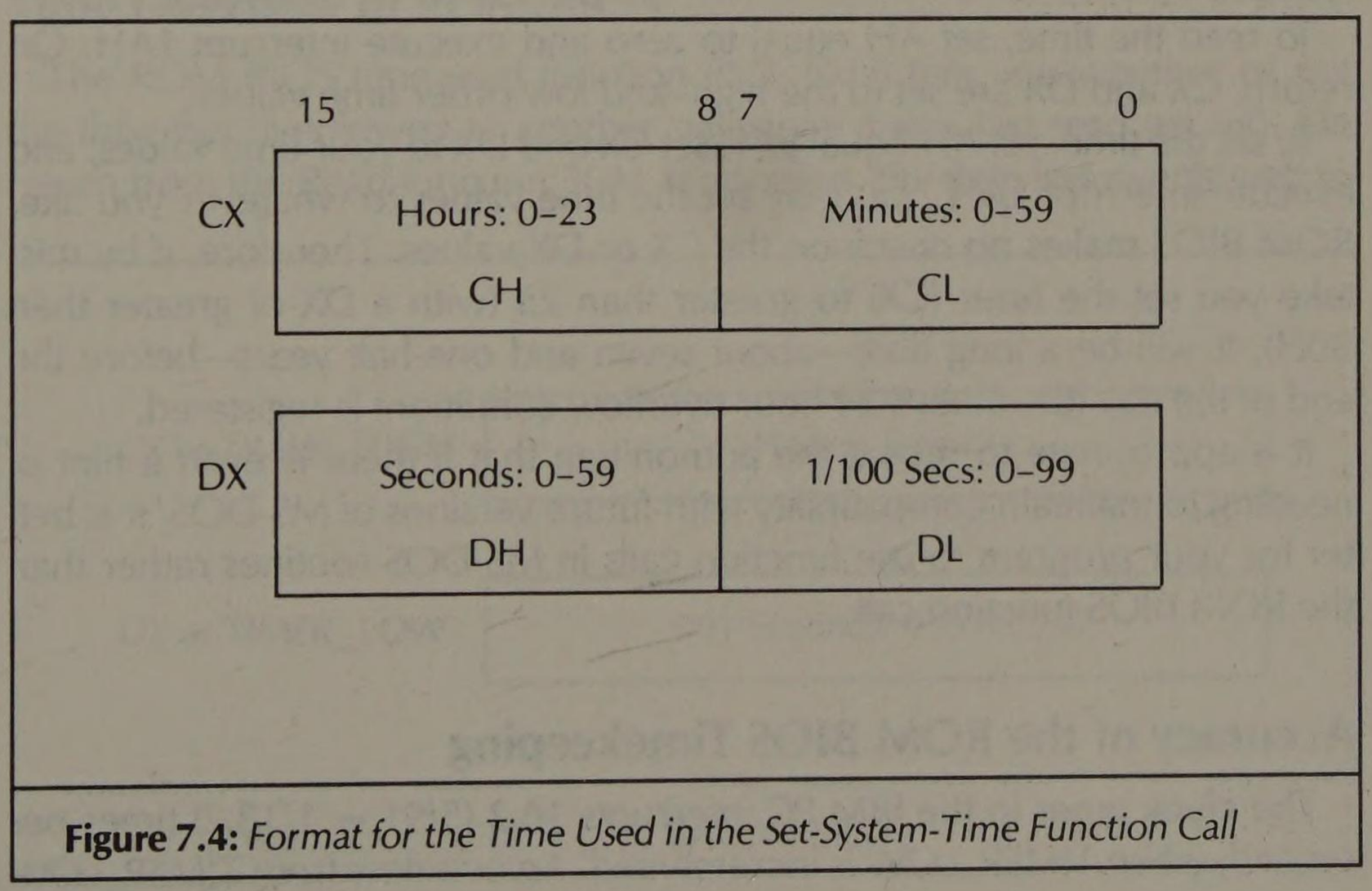


Figure 7.4: *Format for the Time Used in the Set-System-Time Function Call*

ROM BIOS TIME

We now know that ROM BIOS keeps time in the IBM PC compatibles. MS-DOS uses interrupt 1A, the time-of-day interrupt call, to get its time value from ROM BIOS. Let's now examine the time format that ROM BIOS uses and discuss the function call that gets and sets the time. After that we will cover the subject of time overflow in ROM BIOS.

Time Format in ROM BIOS

ROM BIOS uses a time format unlike those used in MS-DOS or the directory entry. The time is kept in two words. The high-order word, TIMER_HIGH, contains the hours, 0–23 (sometimes higher than 23); the low-order word, TIMER_LOW, contains a count of increments of approximately 5/91 seconds. The sanity of choosing 5/91 seconds as a unit becomes apparent when you see that 64K (65,536) of these units make up 3600 seconds, or one hour. TIMER_HIGH and TIMER_LOW are the labels of the locations that ROM BIOS uses to store the time.

The ROM BIOS Read- and Set-System-Time Function Call

Interrupt 1A calls the ROM BIOS functions that read and set the time of day. CX and DX are used for passing the arguments: the high-order value is in CX, the low-order value in DX. CX contains the hours; DX contains the number of seconds × 18.206481. Figure 7.5 shows the ROM BIOS time formats.

To read the time, set AH equal to zero and execute interrupt 1AH. On return, CX and DX are set to the high- and low-order time values.

To set the time, set AH equal to 1, set CX and DX to your time values, and execute interrupt 1AH. You may set the time values to whatever you like. ROM BIOS makes no check on the CX or DX values. Therefore, if by mistake you set the hour (CX) to greater than 23 (with a DX of greater than B0H), it will be a long time—about seven and one-half years—before the end of the day (the timer's 24-hour overflow condition) is registered.

It is appropriate to repeat the admonition that if there is even a hint of needing to maintain compatibility with future versions of MS-DOS, it is better for your program to use function calls in MS-DOS routines rather than the ROM BIOS function call.

Accuracy of the ROM BIOS Timekeeping

The clock timer in the IBM PC interrupts 18.2 (5/91 = 1/18.2) times per second, when TIMER_LOW is incremented. An overflow from TIMER_LOW

increments TIMER_HIGH at almost exactly one-hour intervals. In an attempt to be as accurate as possible, the ROM BIOS routines adjust for exactness at the end of every day by waiting an additional 9.65 seconds before setting the time to zero for the next day. This suggests that the COMPAQ clock gains just under one-half second every hour, but it corrects for that at the end of each day.

How accurately 18.2 times per second describes the time interval between interrupts can be seen by reviewing the following information: there are 1,193,180/65,536 or 18.206481 interrupts per second; thus, each interval is 0.0549254 seconds.

Computing Time Differences in ROM BIOS Times

Finding the difference between two times stored in ROM BIOS format is truly simple: you subtract the smaller from the larger.

The Location of the Time Cells

Now that you understand how timekeeping works, you may want to see the two timer locations: TIMER_LOW and TIMER_HIGH. You can use DEBUG to look at locations 0:046C and 0:046E. It appeals to some to look at MS-DOS busily working when it appears to be quiet. Given a highly anthropomorphic, and perhaps maudlin, point of view, 0:046C is evidence of a COMPAQ heartbeat.

Time Overflow in ROM BIOS

The ROM BIOS time-read function (INT 1AH) tells you whether or not the time has overflowed to another day since it was last read (or set). On return from the Read function, if AL is nonzero, the time has overflowed to

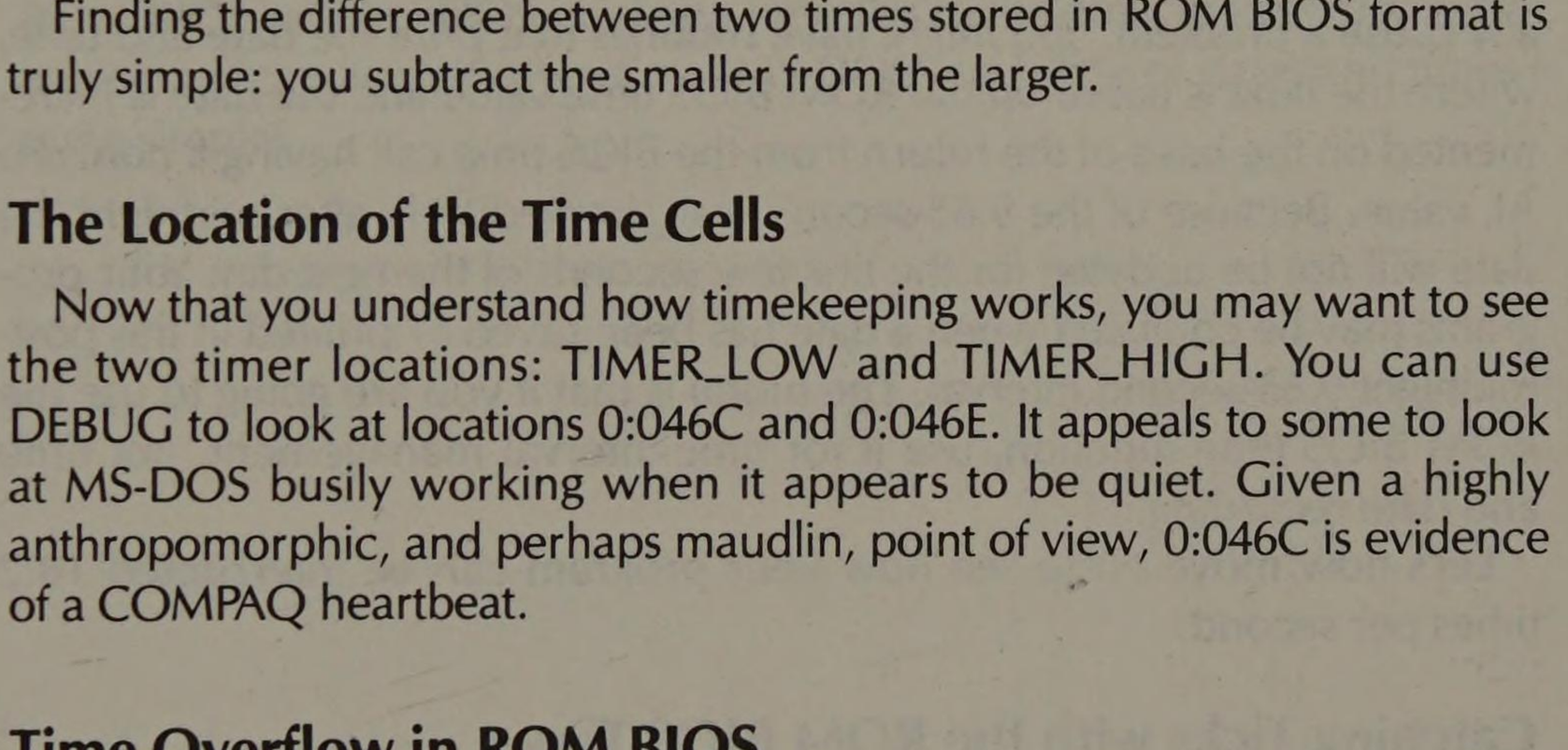


Figure 7.5: *Format for the Time Used by ROM BIOS*

another day. If AL is zero, the timer is still on the same day as when it was last read or set. (Note: AL does not contain the number of days the time has overflowed.)

We have previously discussed the (approximate) half-second-per-hour "gain" in the ROM BIOS clock. We also showed how this is compensated for once a day (the time-update routines make the transition from one day to another when TIMER_HIGH equals 24 *and* TIMER_LOW is equal to B0H). The time overflow flag is not set until the extra time (9.65 seconds) has elapsed. The next day's time is established by setting both TIMER_HIGH and TIMER_LOW to zero. (Note: This adjustment is visible if your program uses the ROM BIOS time function call; it will not be visible if it uses the MS-DOS function calls.)

This review of date updating points us to a situation that might occasionally cause a problem. You might have routines that print the date and time, where the time is based on the ROM BIOS time value and the date is incremented on the basis of the return from the BIOS time call having a nonzero AL value. Because of the 9.65-second wait, immediately after midnight the date will not be updated for the first few seconds of the next day. Your programs may be confused when a date has been saved or printed in this post-midnight 9.65-second interval. The moral is that if you are going to use the ROM BIOS time function, use it for time-interval management, not time and date recording.

Let's now move on to see how your program can be interrupted 18.2 times per second.

Catching Ticks with the ROM BIOS Timer

There is a feature in the IBM PCs compatibles' ROM BIOS that will interrupt a program 18.2 times per second. Interrupt 1CH is executed every time the ROM BIOS timer routine is executed. This means that you don't have to use the ROM BIOS time values to measure elapsed time; your program can keep track of time in its own way.

On system restart, the vector for interrupt 1CH (i.e., bytes 70 to 73H) is initialized to point to an IRET instruction. Your program can redirect the vector to your own timer-tick service routine. With this routine you can keep track of time in a format that is more convenient than that provided by ROM BIOS or MS-DOS. But don't forget that before your program terminates, it must reset the interrupt vector to its original value.

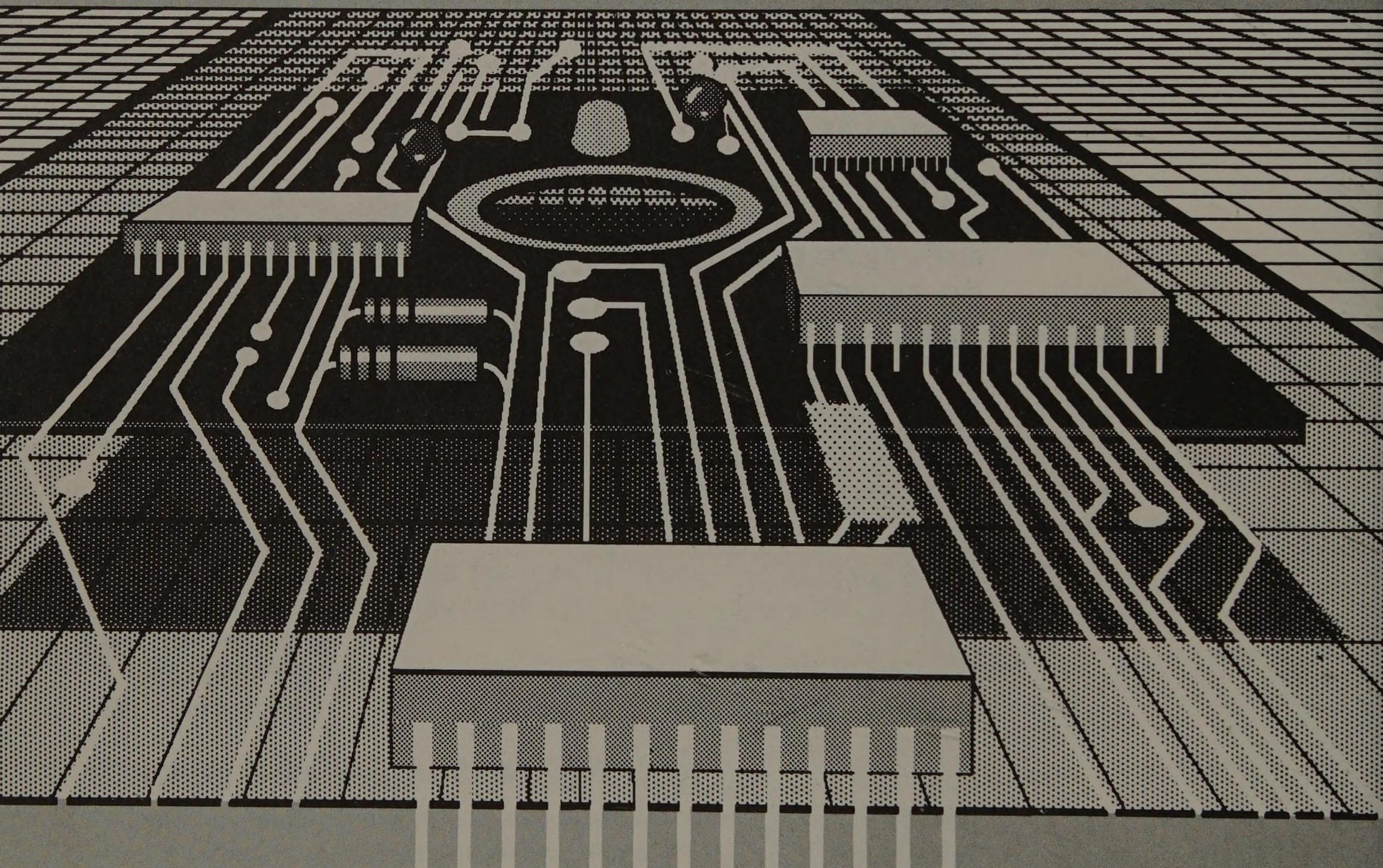
TOTAL TIME CONTROL

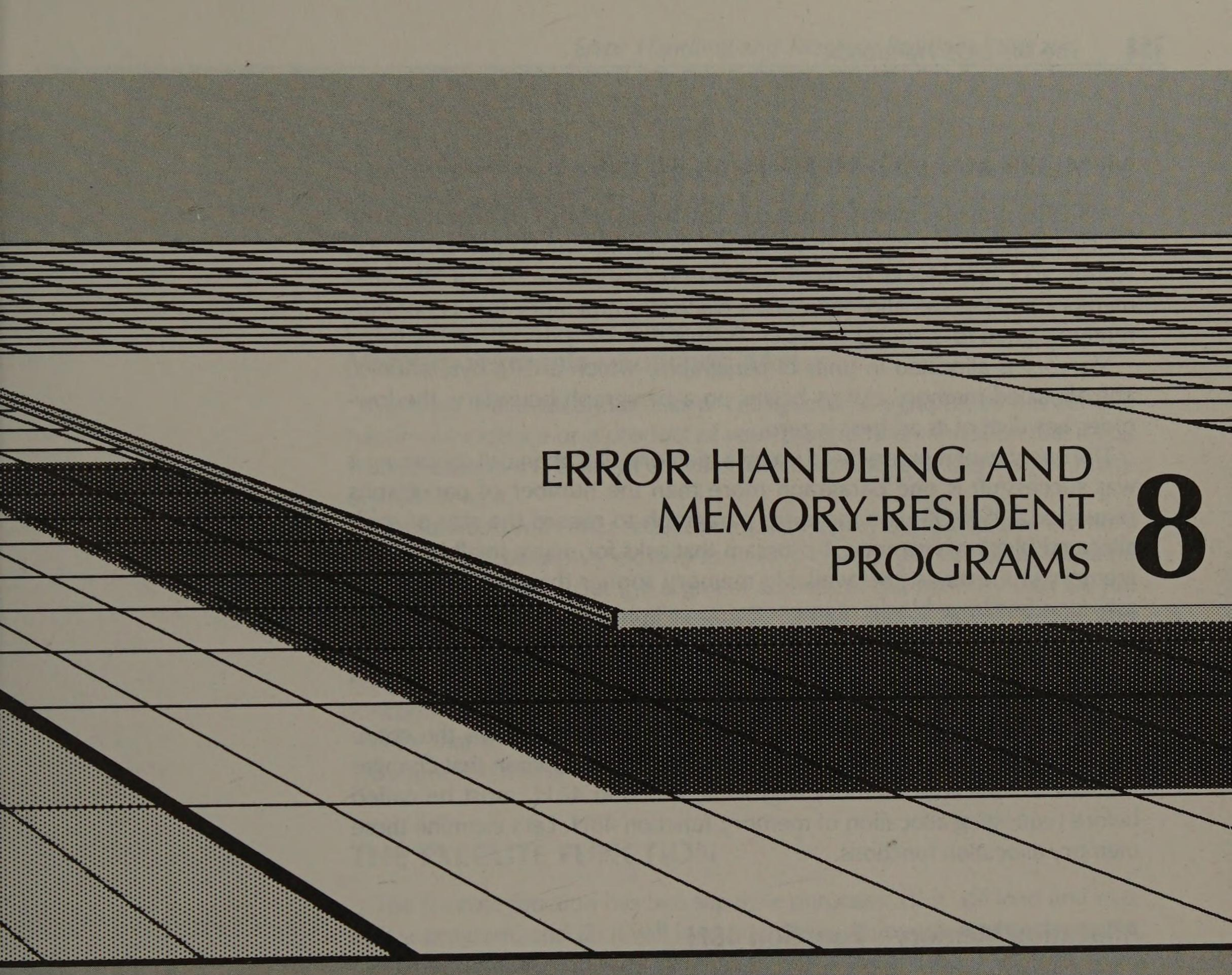
Your software can take over the entire management of time by redirecting interrupt 8H, locations 20 to 23H. The ROM BIOS time-update routines

will no longer be executed, and all time-management functions, such as providing a response to the MS-DOS read- and set-system-time function calls, become your program's responsibility. This may turn out to be a heavy burden with few advantages—power with a price.

SUMMARY

In this chapter, we have learned about the two MS-DOS date formats; the one that is used in directory entries, and the other that is used in the date-access function calls. We have also seen three time formats: one used in directory entries, one used in the MS-DOS time-access function calls, and one used by ROM BIOS. We examined the accuracy of timekeeping in both ROM BIOS and MS-DOS, and saw how, by redirecting an interrupt vector, a program can keep its own time. MS-DOS's handling of dates and times is quite simple.





ERROR HANDLING AND MEMORY-RESIDENT PROGRAMS

8

INTRODUCTION

This chapter discusses an important feature of MS-DOS that contributes so much to its usefulness: the ability to execute additional memory-resident programs. Without this ability, programs such as Sidekick and ProKey would not be available. The chapter examines the tools that manage the memory and the tools that execute, and sometimes wait for completion of, other resident processes.

We also examine function call error reporting. The level of error condition reporting has improved with each succeeding major release of MS-DOS, culminating in version 3 with a function designed especially to help the programmer deal with error conditions.

MEMORY MANAGEMENT FUNCTIONS

MS-DOS provides memory management functions that (1) allocate to the caller a block of memory, (2) deallocate (return to the system pool) previously allocated blocks of memory, and (3) change the size of allocated memory blocks. The allocation takes place from memory that is not currently assigned to other purposes.

Memory is allocated in units of *paragraphs,* which are 16-byte chunks. The allocated memory always begins on a paragraph boundary; the low-order hex digit of its address is zero.

The quantity of memory used up by a memory allocation call (assuming it was successful) is one paragraph more than the number of paragraphs requested. MS-DOS uses the extra paragraph to record the size of each allocated block of memory. A program that asks for many small blocks of memory will exhaust the available memory sooner than a program that asks for a few large blocks of memory.

If you ask for two blocks of memory in a row, the blocks will not be contiguous. They will be separated by the record-keeping paragraph that belongs to the second block of allocated memory.

The MS-DOS command processor assigns to most programs the entire memory pool when the programs are loaded, so the function that changes the size of allocated memory segments, function 4AH, must be called before requesting allocation of memory, function 48H. Let's examine these memory allocation functions.

Allocating Memory—Function 48H

This memory-allocation function allocates the number of paragraphs specified in BX. It returns in AX the segment address of the allocated memory.

To acquire memory, put the number of paragraphs required in BX, set AH equal to 48H, and execute interrupt 21H. On return, if the memory was available, AX:0 points to the allocated memory. If not enough memory was available, the carry bit will be set and BX will contain the maximum number of paragraphs that could be successfully allocated.

Deallocating Memory—Function 49H

This function returns, to the system pool, memory that has been previously allocated by a call to function 48H. The function will fail if the memory control block (the extra paragraph) associated with the previously allocated memory has been damaged, or if the deallocation function does not recognize the address of the block being returned.

To return memory to the system pool, put the segment address of the allocated block into ES, set AH equal to 49H, and execute interrupt 21H. If the carry bit is set on return, the function has failed for one of the reasons just mentioned.

Changing the Size of Allocated Memory Segments—Function 4AH

Memory is allocated in blocks of contiguous paragraphs, so it is either a happy coincidence or a product of your program's organization that if you wish to expand a previously allocated memory segment, there is free memory following it.

This function will either reduce the size of a previously allocated memory block or, if space allows, expand its size. To change the size of an allocated segment of memory, put the segment address of the memory into ES, put the new number of paragraphs requested in BX, set AH equal to 4AH, and execute interrupt 21H. On return, if the carry bit is set, BX will contain the maximum number of paragraphs available for allocation, as it does when function 48H fails.

Now that we have seen the memory management functions, let's examine a function that is often used along with them, the Execute function.

THE EXECUTE FUNCTION

The Execute function has two separate purposes: (1) it will load and execute a program, and (2) it will load an overlay. Using this function requires more preparation than most other functions require. Once mastered, however, the Execute function is powerful and useful.

Before we see how to use the Execute function, let's have a look at some of the implications of its existence.

The Execute Function and Multitasking

Once a program has called the Execute function, it can do nothing else until the program it called is complete (this would not be true in a multitasking system). So, typically, the calling program (the one that calls the Execute function) will then immediately call the Wait function. The Wait function (4DH) gets the return code supplied by the called program.

If MS-DOS made multitasking available to us, our programs could keep working (i.e., they would not immediately call the Wait function) while the called program was being executed. We could also start dealing with processes as well as programs, and really keep the 8086/8088 busy. MS-DOS is tantalizingly close to multitasking.

Executing MS-DOS Commands

The program that is loaded by the Execute function may be another copy of the command processor. Thus, all the normal MS-DOS commands are, in effect, at the disposal of any program. See Chapter 14, "Batch Operations," for an example of how the Execute function is involved in calling a batch file as a subroutine.

Using the Load and Execute Option of Function 4B

Let's have a look at the steps required to use the load-and-execute option. They are as follows:

1. Make sure memory space is available.
2. Build a parameter block.
3. Build a drive, path, and program-name string.
4. Call the Execute function.
5. Call the Wait function.

We will examine these steps one by one, presenting an example at the end.

Returning Memory to the Pool

In the current versions of MS-DOS, programs usually have allocated to them the entire installed contiguous memory of the computer. Other programs cannot be loaded until some of this memory is returned to the system's pool of available memory. This is accomplished with function 4AH, which takes as an argument the segment address of the allocated memory block.

To return memory to the system pool, you need two pieces of information: (1) the length, in paragraphs, of the currently running program (so that you know the new memory block size to give function 4AH), and (2) the segment address of the currently running program.

When the currently running program was loaded, the memory-management function was used to get the space to hold it. The segment address of the allocated memory was put into the CS, DS, and ES registers.

Constructing the Parameter Block

Once enough space is available for the program to be executed, a parameter block needs to be constructed so that the called program will have (1) an environment string, (2) a command line (at 80H in the new PSP), and (3) correctly set default file control blocks at 5C and 6C in the new program segment prefix. The formats of the parameter blocks, for loading and executing a program, and for loading an overlay, are shown in Figures 8.1 and 8.2.

The Environment String You can create your own, or you can use the environment string belonging to the currently running process (the program calling the Execute function). Creating your own requires an understanding of the contents of the CONFIG.SYS file. To use the existing environment string, set the first two bytes of the parameter block to zero. Location 2CH in the program segment prefix shows the address of the existing environment.

Creating a Command Line Your program needs to create the command line that will be moved to 80H of the called program. The format of the command line is

L string CR

where *L* is the number of bytes in the command text, *string* is the text of the command, and *CR* is 0DH. Note that the command text does not contain the name of the program being called, just the characters that would follow the command, including the space and slash characters as appropriate.

The Default File Control Blocks If your program is using only file handle file-access functions, it does not matter what the default FCBs contain. Bytes 6–13 of the parameter table may be 0. However, for version 1 file access, if

Format	Content
WORD	Segment address of the environment
DWORD	Pointer to command line at 80H
DWORD	Pointer to default FCB to be passed at 5C
DWORD	Pointer to default FCB to be passed at 6C

Figure 8.1: *EXEC Parameter Block for Loading and Executing a Program*

Format	Content
WORD	Segment address where file will be loaded
WORD	Relocation factor to be applied to the image

Figure 8.2: *EXEC Parameter Block for Loading an Overlay*

the command line's first and second arguments are file names, then bytes 6–9 and bytes 10–13 of the parameter block should point to default file control blocks.

The default file control block is 16 bytes long and consists of (1) a one-byte drive designator (00 for default drive, 01 for drive A, 02 for drive B, etc.), followed by (2) the filename of the first or second argument, in uppercase, padded out with spaces on the right, if necessary, to make it eight characters long, and (3) the filename extension in uppercase, padded with spaces on the right, if necessary, to make it three characters long, and (4) four undefined bytes.

Bytes 6–9 of the parameter blocks should point to the default FCB constructed from the first file name argument.

Building a Drive, Path, and Command Name String

Your program needs to contain a string of the form

d:\path\file name

terminated by a zero byte. This is the string that tells the Execute function which program to execute.

Calling the Execute Function

To execute another program, point DS:DX to the ASCII drive, path, and file name string (which is terminated by a zero byte, as always), point ES:BX to the constructed parameter block, set AL equal to 0 and AH equal to 4BH, save the registers that your program cares about on the stack, save the SS and SP registers other than on the stack, and execute interrupt 21H.

On return from the executed function, most of the registers have been changed, including SS and SP. At this point, SS and SP must be restored from where they were saved, and the registers your program cares about must be restored from the stack.

Calling the Wait Function

The calling program stops executing until the called program runs to completion (or is terminated early for some reason). (In versions that include multitasking, the calling program could continue to run until it needs to know that the called program has concluded.) The calling program uses the Wait function to wait for the completion of the called program, and to collect whatever error or completion code is returned as the called program.

The executed program uses function 4CH to return its error or completion code. This code is available to the calling program when it calls the Wait function, 4DH. A program can only call the Wait function once for

each program it executes; a second Wait call would risk waiting for the completion of a never-executed program.

An Example

This example shows how a program can execute any of the regular MS-DOS commands. The difference between this example and one in which a program of your own creation is called will occur in the name of the program called and the command line passed. Let's assume that you want a program to run CHKDSK on the disk in drive B. We will also assume that the system's memory pool is adequate to hold a copy of the command processor (COMMAND.COM), which is the program we are going to execute.

Parameter Block Construction We will pass on the existing environment, so word zero (the first two bytes) of the parameter block is zero.

The command line to be placed at 81H in the called program's PSP will look like this:

```
/C CHKDSK B:
```

At 80H will be the count of characters in the string (0CH). The character following the string (the terminator) will be 0DH. Thus, words one and two will point to an area whose first byte is 0CH, whose following bytes contain /C CHKDSK B:, and whose last byte contains 0DH. Word one is the low-order (offset) part of the address; word two is the high-order (segment) part of the address.

The command processor does not use the formatted file control blocks at 5C and 6C to run the CHKDSK command, so we can set the next two pairs of words to point to a character string of eleven spaces.

Drive, Path, and Command-Name String The program we wish to execute is COMMAND.COM. It resides on drive A in the primary directory, so the string that DS:DX will point to is

```
a:command.com
```

terminated by a zero byte. We have now set up the parameters that the Execute function will use. It is time to call it.

We might think that CHKDSK.COM could be directly executed. This is not done without difficulty, because the command processor and CHKDSK.COM know about one another and have their own method of communicating arguments. That's why we had COMMAND.COM do the work for us.

Now we will switch subjects and examine the function call error reporting facilities of MS-DOS. With the introduction of networking, more extensive function call error reporting becomes necessary.

FUNCTION CALL ERROR REPORTING

When a function recognizes that it cannot do what was asked of it, it returns an error condition. It is up to the calling program to take the right action. The conventions for error reporting in MS-DOS have been improved with the release of each new major version.

In version 1 there was no approach to error reporting that was consistent throughout the functions. Often if the AL register was equal to FFH on return, or just not equal to zero, that indicated an error condition. However, some functions required a unique test to be made to determine if an error had occurred. While there existed in version 1 relatively few functions, the lack of an error reporting convention was not too painful to work with.

Version 2 introduced the convention that if an error condition existed, the C bit would be set on return from the function; otherwise the C bit would be clear. The error type, returned in AX, could be used as an index into a list of 17 different error conditions. The action to take on encountering these conditions could be derived from reading the *MS-DOS Technical Reference Manual* (usually available in the form of the *IBM Technical Reference*), from experimenting with the functions, or not at all.

Version 3 has moved a major step forward. Part of the propulsive force has been the arrival of the networking capability, where error conditions are used to indicate transitory states, not just permanent faults. The action to take on encountering an error is much more context-dependent than in previous versions. In version 3, when functions detect an error your program calls function 59H—Get Extended Error—to find out more about the error.

Get Extended Error—Function 59H

New programs, especially those intended to run on a network, must call this function when other functions indicate an error condition. Not all functions can make use of this extended error reporting. The functions that can are interrupt 21H functions that return the C bit to indicate an error, interrupt 21H functions involving file control blocks that return FFH to indicate an error, and interrupt 24H function calls.

Function 59H returns four separate items of information: the error code, the error class, the locus, and the suggested action.

Error Code

The error code is returned in AX. See Figure 8.3 for a list of the error codes. Codes 1 through 18 match those generated by the version 2 function calls that use the C bit to indicate an error condition. Codes 19 through 31 correspond to the codes 0 through 12 that are returned by interrupt

24H in version 2. Codes 32 and above are generated by version 3 functions; they refer to file sharing and other network-related activities.

Error Code Dec (Hex)	Meaning
1 (01H)	Function number not valid
2 (02H)	File not found
3 (03H)	Path not found
4 (04H)	Too many open files (no handles left)
5 (05H)	Access denied (file not open for type of access requested, or other condition)
6 (06H)	Invalid handle
7 (07H)	Memory control block destroyed
8 (08H)	Insufficient memory
9 (09H)	Invalid memory block address
10 (0AH)	Invalid environment
11 (0BH)	Invalid format
12 (0CH)	Invalid access code
13 (0DH)	Invalid data
14 (0EH)	Reserved
15 (0FH)	Invalid drive was specified
16 (10H)	Attempt to remove the current directory
17 (11H)	Not the same device
18 (12H)	No more files
19 (13H)	Attempt to write on write-protected disk
20 (14H)	Unknown unit
21 (15H)	Drive not ready
22 (16H)	Unknown command
23 (17H)	Data error (CRC)
24 (18H)	Bad request structure length
25 (19H)	Seek error
26 (1AH)	Unknown media type
27 (1BH)	Sector not found
28 (1CH)	Printer out of paper

Figure 8.3: *Error Codes Returned by Function 59H*

Error Code Dec (Hex)	Meaning
29 (1DH)	Write fault
30 (1EH)	Read fault
31 (1FH)	General failure
32 (20H)	Sharing violation
33 (21H)	Lock violation
34 (22H)	Invalid disk change
35 (23H)	FCB not available
36 (24H)	Sharing buffer overflow
37–49 (25H–31H)	Reserved
50 (32H)	Network request not supported
51 (33H)	Remote computer not listening
52 (34H)	Duplicate name on network
53 (35H)	Network name not found
54 (36H)	Network busy
55 (37H)	Network device no longer exists
56 (38H)	Net BIOS command limit exceeded
57 (39H)	Network adaptor hardware failure
58 (3AH)	Incorrect response from network
59 (3BH)	Unexpected network error
60 (3CH)	Incompatible remote adaptor
61 (3DH)	Print queue full
62 (3EH)	Not enough space for print file
63 (3FH)	Print file deleted
64 (40H)	Network name deleted
65 (41H)	Access denied
66 (42H)	Network device type incorrect
67 (43H)	Network name not found
68 (44H)	Network name limit exceeded
69 (45H)	Net BIOS session limit exceeded
70 (46H)	Temporarily paused
71 (47H)	Network request not accepted

Figure 8.3: *Error Codes Returned by Function 59H (continued)*

Error Code Dec (Hex)	Meaning
72 (48H)	Print or disk redirection is paused
73–79 (49H–4FH)	Reserved
80 (50H)	File exists
81 (51H)	Reserved
82 (52H)	Cannot make directory entry
83 (53H)	Fail on INT 24H
84 (54H)	Too many redirections
85 (55H)	Duplicate redirection
86 (56H)	Invalid password
87 (57H)	Invalid parameter
88 (58H)	Network device fault

Figure 8.3: *Error Codes Returned by Function 59H (continued)*

It may be clear what your program should do when it encounters one of these errors. If not, the other three items returned by function 59H might be helpful.

Error Class

The error class is returned in register BH. It tells you what sort of error was encountered; this gives the program an indication of what to do about the error, for example, whether to wait and try again or to accept that all is lost. The error classes are:

1 Resource exhausted; e.g., out of space
2 Temporary situation; e.g., a locked file
3 Authorization problem; e.g., lacking permission
4 Internal system (i.e., MS-DOS) software error
5 Hardware failure
6 System failure; e.g., missing CONFIG.SYS file
7 Inconsistent requests
8 Item or file not found
9 Incorrect format for item or file
10 Locked item or file

11 Media failure; e.g., CRC error on disk
12 Item, e.g., machine name or directory, already exists
13 Not known

Classes 4, 5, and 6 indicate situations that are not the fault of your program. Class 7 needs your attention. Once the error class is known, your program may have a clear path toward dealing with the error. If not, the locus will help.

Locus

The locus is returned in register CH. It tells you what section or area of the system is involved. The returned value is between 1 and 5. Value 1 is used when function 59H has nothing specific to say about the locus. The meanings of the other locus values are:

2 Mass storage, usually disk
3 Network
4 Serial device
5 Memory

Given the error code and the locus, your program will normally know enough to take appropriate action when it encounters an error. However, that is not always true, and help is at hand.

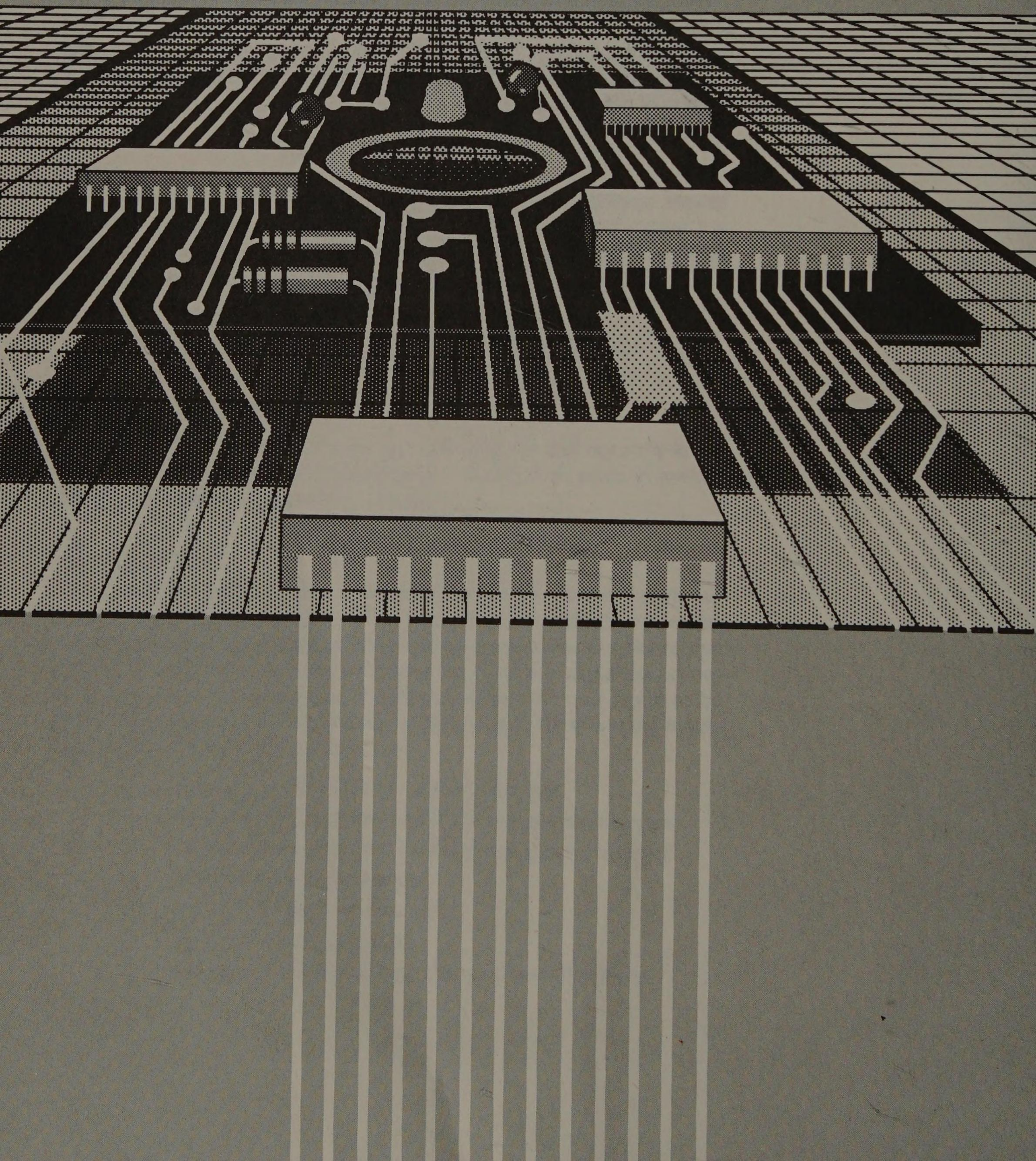
Suggested Action

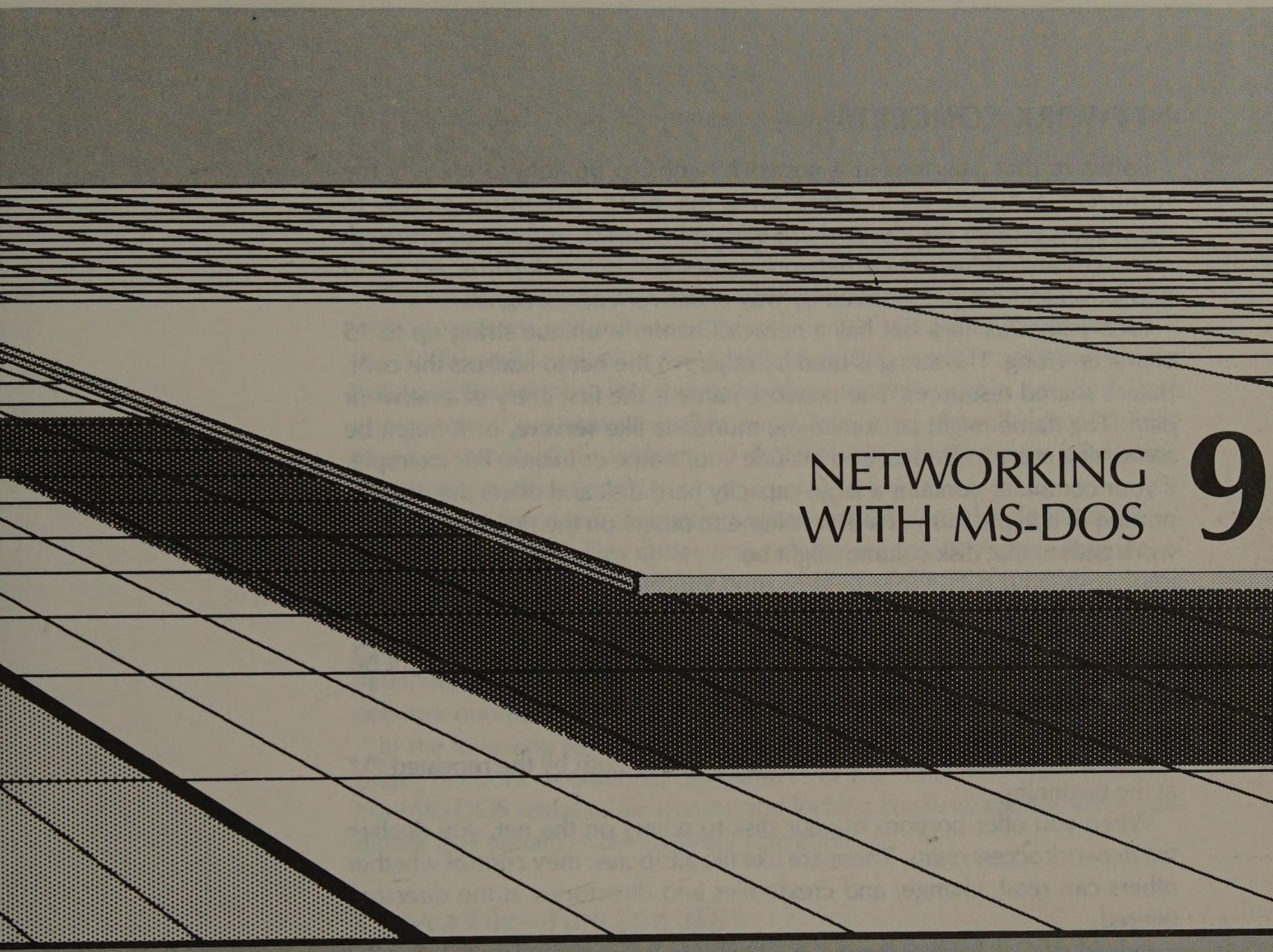
The suggested action code is returned in register BL. Your program can use this information if the error is not anticipated. In addition, reviewing these suggested actions now can indirectly alert you to the conditions that have to be anticipated by your programs. Some of the suggested actions involve the user when the error condition persists. The user needs to be asked if the program should continue or be aborted. These are the meanings of the suggested action code values:

1 Retry a few times. If error persists, ask user.
2 Pause then retry (a few times). If error persists, ask user.
3 Ask user to reenter item (unless the program created it).
4 Abort with orderly cleanup.
5 Immediate abort, no cleanup.
6 Ignore this error condition.
7 Ask user to intervene (e.g., change disks), then retry.

SUMMARY

We have seen how more than one program can exist in memory at the same time, and how one of those programs can call another. We have examined the different levels of error reporting supplied by MS-DOS, and seen how extended error handling has become necessary with the introduction of networking.





NETWORKING WITH MS-DOS 9

INTRODUCTION

MS-DOS, beginning with version 3.1, has within it a limited set of functions for network support. You can specify the conditions under which files and directories are shared, you can link to directories and devices (printers) that belong to other computers on the net, and you can lock and unlock individual portions of files. To manage the network operations additional software, such as the PC Network program, is required.

This chapter notes the concepts involved when MS-DOS is part of a network, shows how existing MS-DOS functions are affected by the presence of network operations, and describes the network-specific functions that control file sharing and linking (alias redirection).

NETWORK CONCEPTS

Software that operates in a network needs to be able to identify the resources available to it; users on a net need to control access to the resources they decide to share with other users in the net. Network resources are addressed via *network names* and *network paths;* access to network resources is controlled by way of *network access rights.*

Each computer in a net has a *network name,* a unique string up to 15 characters long. This string is used by others in the net to address the computer's shared resources. The network name is the first entry in a *network path.* The name might be something mundane like **serverc,** or it might be something more individual and include your name or initials. For example, if your computer contains a large-capacity hard disk and offers the use of a portion of it in the form of a disk volume to others on the net, then the network path to that disk volume might be

```
\\rakserver\netdisk
```

The network path to a printer attached to a central printer server might be

```
\\pserver\LPT1
```

A network path is distinguished from an ordinary path by the repeated "\" at the beginning.

When you offer portions of your disk to others on the net, you declare the *network access rights.* These are like file attributes; they control whether others can read, change, and create files and directories in the directory offered.

Individual files have a *sharing mode* that is declared when the files are opened using functions 3CH and 5BH. The sharing mode provides control over access of other processes to individual files, like the network access rights provide control in an offered directory.

For file sharing to operate, the SHARE command must have been called prior to opening any files whose sharing mode you wish to control. The SHARE command needs to be called only once each time you boot the system.

Joining the Net

Network operations are invoked by calling the network program from the usual MS-DOS prompt, after the SHARE command has been executed. When you join the net, you specify which directories and devices (printers) of yours you are offering for sharing. At that time you create network access rights to those devices/directories. After a few times of running through these specifications, you will generate an addition to your

AUTOEXEC.BAT file to automatically initialize your computer for network operations.

MS-DOS FUNCTIONS UNDER NETWORK OPERATIONS

The major difference between software that runs under MS-DOS network operations and software that runs under, say, MS-DOS version 2.1 is the additional level of error possibilities that needs managing under network operations. For example, the response to file access failure under network operations might be to retry after a small delay—the file may be temporarily locked. In non-network systems, file access failure indicates a more serious error.

Another difference is that your program needs to use the file handle functions for controlling files under network operations; any interesting use of the net requires file handle functions instead of the FCB functions. Some minor, but significant, differences exist in the way some functions are called. All MS-DOS functions except open file and I/O control (functions 3DH and 44H) take the same arguments whether they are running under network operations or not.

In the following sections we describe the file opening function as it runs under network operations, the features of the IOCTL function that alter how MS-DOS reacts to file sharing and locking conflicts, and the error conditions that pertain under network operations.

Open a File—Function 3DH

Function 3DH takes two arguments: in DS:DX a pointer to an ASCIIZ string that identifies the file to be opened, and in AL a mode in which the file is to be opened. Before the advent of network operations (i.e., in version 2), the AL argument is called the *access code*. This access code defines whether the file is opened for reading, for writing, or for both reading and writing.

Under network operations, for which version 3.1 or later is required, the AL argument becomes the *open mode*. The number of significant bits in the AL argument expands from 2 to 8. The low order three bits are devoted to the original access code specification; the additional bits control the sharing mode and an inheritance flag.

The Sharing Mode

The sharing mode is specified in bits 4, 5, and 6 of the open mode argument in AL. It defines the level to which you are willing to have other processes interact with your file. It's a negative indicator, in that the values taken

by the sharing mode deny, rather than permit, forms of access. The sharing mode values are

0	Compatibility
1	Deny read/write
2	Deny write
3	Deny read
4	Deny none

The most common sharing mode for network operations is deny write. The file may be read by any process, but not modified (written). Sharing mode 2 accomplishes this protection.

If you run programs written for version 2 under a network, the access code specifies, by default, a sharing mode of 0. This is compatibility mode and denies any access by other processes to the files opened by version 2 software. Version 2 software does not (in general) anticipate network operations, and so version 3's sharing mode 0 provides compatibility with previous versions of MS-DOS.

Sharing mode 4, deny none, opens the file to both reading and writing by other processes. This is quite adventurous. It is appropriate only if all the other software operating on the network uses properly executed file-locking protocols.

When your program attempts to open a file on, say, another computer's disk on the net, that file may already be open. If the file is open and your program chose the default access code (read/write), your open function call will succeed only if

- the network access rights established when the other computer on the net offered to share its resources include the read/write right, and
- the sharing mode specified by the other process's open function is sharing mode 4, deny none.

In networking situations you need to be aware of the possibility that other processes may already be working with, or may be about to work with, the file you are about to open.

The Inheritance Flag

The inheritance flag, bit 7, is relevant for multiprocessing. The flag can be set to keep the file private to the current process. If the inheritance flag is zero the file can be inherited by child processes (see the discussion of the EXEC function in Chapter 8).

IOCTL—Function 44H

Two subfunctions of function 44H allow programs to find out whether they are dealing with remote or local devices and files, and another

subfunction allows control over the sharing retry count, the semi-automatic mechanism that is designed to wait for locked network files to become unlocked. Programs should make no distinction between local and remote devices, but here's how if you are curious.

Testing for Remoteness To see if a handle is for a local or remote file, place the handle in BX, set AL equal to 09H, set AH equal to 44H, and execute interrupt 21H. Assuming redirection is not paused, if bit 12 in DX is set, the file is remote.

To find out the same information about a device, use the same call except for setting AL equal to 0AH. On return, examine bit 15. If it's set, the device is remote.

Sharing/Locking Conflict Retries Access to a file may be denied because of sharing or locking conflicts. MS-DOS takes a standard approach in this situation; it waits a while, then retries. If that access fails, the wait/retry sequence is repeated until the retry count is exhausted. The default value for the wait is once through the instruction sequence

```
xor CX,CX
loop $
```

and three retries. Subfunction 0BH in function 44H allows you to set the number of times to execute the delay loop before each retry, and the number of retries.

To set the retry and delay counts, set the retry count into DX, the delay count into CX, set AL equal to 0BH, AX equal to 44H, and execute interrupt 21H.

Error Conditions

Under network operations, the treatment of errors returned by some file access functions becomes slightly more complex. File access failure may occur because, for example, the file is locked. Even though MS-DOS retries automatically, the appropriate action for your program to take may be to retry again after a short delay. The alternative of abandoning the attempt to access the file may be unacceptable. (Note: Under networking everything hangs together on the basis that files are locked only briefly.)

File access may also be denied based on the lack of a suitable network access right. Functions that create and delete files and directories need the Create access right; functions that read and write files may come into conflict with the existing network access rights. The rights may vary from time to time, depending on how another user sets up access rights when he or she joins the network.

The following functions are affected by network access rights:

- 13H through 17H, 21H and 22H, 27H, and 28H—the file functions based on file control blocks.
- 39H, 3AH, 3CH, 3DH, 3FH, 40H, 41H, 43H, 56H, 5AH, and 5BH—the file access functions based on file handles or on ASCIIZ strings containing a path and file name.

Programs written to operate under networking can use function 59H—Get Extended Error—to determine the cause of file access failure, and so decide what action to take. (See Chapter 8 for a list of the extended errors that may be reported.)

NETWORK-SPECIFIC FUNCTIONS IN MS-DOS

The MS-DOS functions that are specific to network operations are network redirection, file locking, and printer control. Network redirection functions establish links with the resources provided by the computers on the net. File locking functions allow for successful sharing of files with other users on the net. The printer setup string functions make it easier to use non-local printers. One other network-specific function is Get Machine Name, which is used by the network program to identify where the program exists. All these functions require that the network program (not part of MS-DOS) be loaded.

Network Redirection

Network redirection allows access to resources (printers or directories) belonging to other computers on the net. You can redirect a local print device (e.g., PRN) to a (remote) printer on the net or a drive letter to a remote directory on the net. You can also create a link between your computer and a remote directory without having to specify a drive letter.

Redirections can be established, reviewed, and canceled using function 5FH. The value of AL at the time the function is called specifies the subfunction. Subfunction 03H establishes redirection for both printers and disk drive letters. Subfunction 02H can be used to review the redirection list. Subfunction 04H cancels a redirection.

Redirecting a Device—Function 5F03H

To redirect a device, set BL equal to 3 to specify device, instead of drive letter, redirection. Point DS:SI to the ASCIIZ name that you want redirected (for example, LPT1:); point ES:DI to the network path of the printer to use instead; set AX equal to 5F03H and execute interrupt 21H.

After device redirection is in place, any output sent to LPT1: will be redirected to the print spooler that manages the printer specified by the network path.

Redirecting a Drive Letter—Function 5F03H (again)

To redirect a disk drive letter, set BL equal to 4; point DS:DI to the ASCIIZ drive letter, colon included (i.e., ASCII drive letter, ASCII colon, 00H terminator); point ES:DI to an ASCIIZ string containing the network path to the directory followed by the ASCIIZ password; set AX equal to 5F03H; and execute interrupt 21H.

The password is required and may be up to eight characters long; it may be null. A network path with password contains two null (00H) characters.

For both of these function calls set CX equal to zero. Also for both functions, if redirection is paused they will return an error. Use function 59H to detect that condition.

If you wish to access several net directories, insert the LASTDRIVE command in the CONFIG.SYS file. The LASTDRIVE command expands beyond E: the number of drive letters acceptable to MS-DOS.

Review Redirection—Function 5F02H

MS-DOS maintains a list of redirections created via function 5F03H. Function 5F02H can be used to step through that list of redirections, using BX as an index.

To get a redirection list entry, set the index into BX, set DS:DI to point to a 128-byte buffer for the local device name or local drive letter, set ES:DI to point to a buffer for the network name, set AX equal to 5F02H, and execute interrupt 21H.

To see all the entries, start with BX equal to zero, and increment it until the end of the list is indicated by error code 18.

Cancel Redirection—Function 5F04H

To cancel redirection, set DS:DI to point to the ASCIIZ device or drive letter, set AX equal to 5F04H, and execute interrupt 21H. The cancellation will not occur if network redirection is paused. Check for that error condition and if true, try again later.

File Locking—Function 5CH

File locking allows portions of files to be locked and unlocked, briefly, to protect them from competing accesses initiated by different processes (other programs on the net).

The area to be locked or unlocked within the file is specified in terms of a 32-bit offset (in CX,DX) and a 32-bit length (in SI,DI). To lock an opened file beginning at addrh,addrl, for lengthh,lengthl, put the file handle in BX; set CX equal to addrh, DX equal to addrl, SI equal to lengthh, and DI equal to lengthl; set AL equal to zero; set AH equal to 5CH; and execute interrupt 21H.

To unlock an already locked area, use exactly the same arguments except set AL equal to 1.

Only one area of a file may be locked at any one time, and it should be locked only briefly. The standard approach taken by programs that find a file on the net unavailable is to retry the operation. (The number of times to retry, and the interval between retries, can be specified via the IOCTL function, 44H.) Network operations are compromised if a program does not unlock a file immediately after, say, the write operation for which it was locked.

Printer Setup Strings

In order to avoid everyone on the net having to use just one print mode, MS-DOS allows a printer setup string to be defined by each user linked to a printer. This printer setup string is sent to the printer ahead of the files that the user prints.

Depending on the flexibility of the printer, setup strings can define the character font, the number of lines per page, the number of characters per line, the page alignment, and other printer-specific features. Each user of a network printer can define a 64-byte string to be sent to the printer.

Define Printer Setup String—Function 5E02H

To specify your setup string for a network printer, set DS:DX to point to the setup string, set the length of the setup string in CX, set the redirection index of the network printer into BX, set AX equal to 5E02H, and execute interrupt 21H.

On return, if C is set, redirection access to the printer is temporarily denied, or no network program is loaded. The error code is returned in AX. The printer setup strings may also be read.

Reading a Printer Setup String—Function 5E03H

To read a printer setup string, you need the redirection index for that printer; it can be found using function 5E02H. Put the redirection index into BX, point ES:DI to a buffer for the string, set AX equal to 5E03H, and execute interrupt 21H. On return, if C is not set, the length of the string in CX and the setup string is in the ES:DI buffer; otherwise, the error code is in AX.

Get Machine Name—Function 5E00H

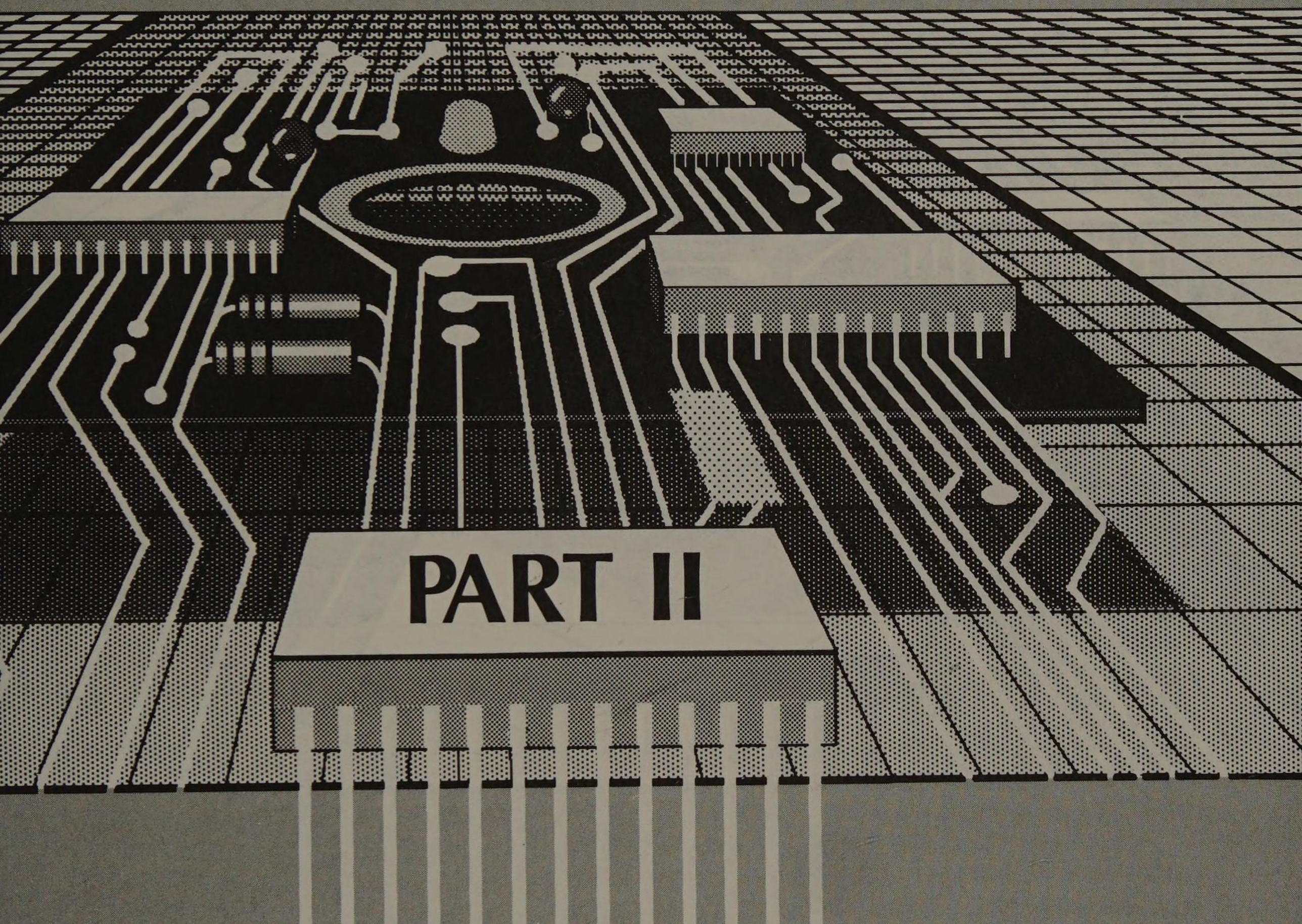
Network software needs to know the name of the device it's running on, and this function provides that name. The network program must be loaded for this function to work correctly.

To get the machine name, set AX to 5E00H, set DS:DX to point to a 16-byte buffer where the name will be put, and call interrupt 21H. On return, if C is set AX contains error codes. Otherwise, if the name has been defined, CH is non-zero; CL contains the network program's name number; and the name, right-filled with spaces and terminated by 00H, is in the DS:DX buffer.

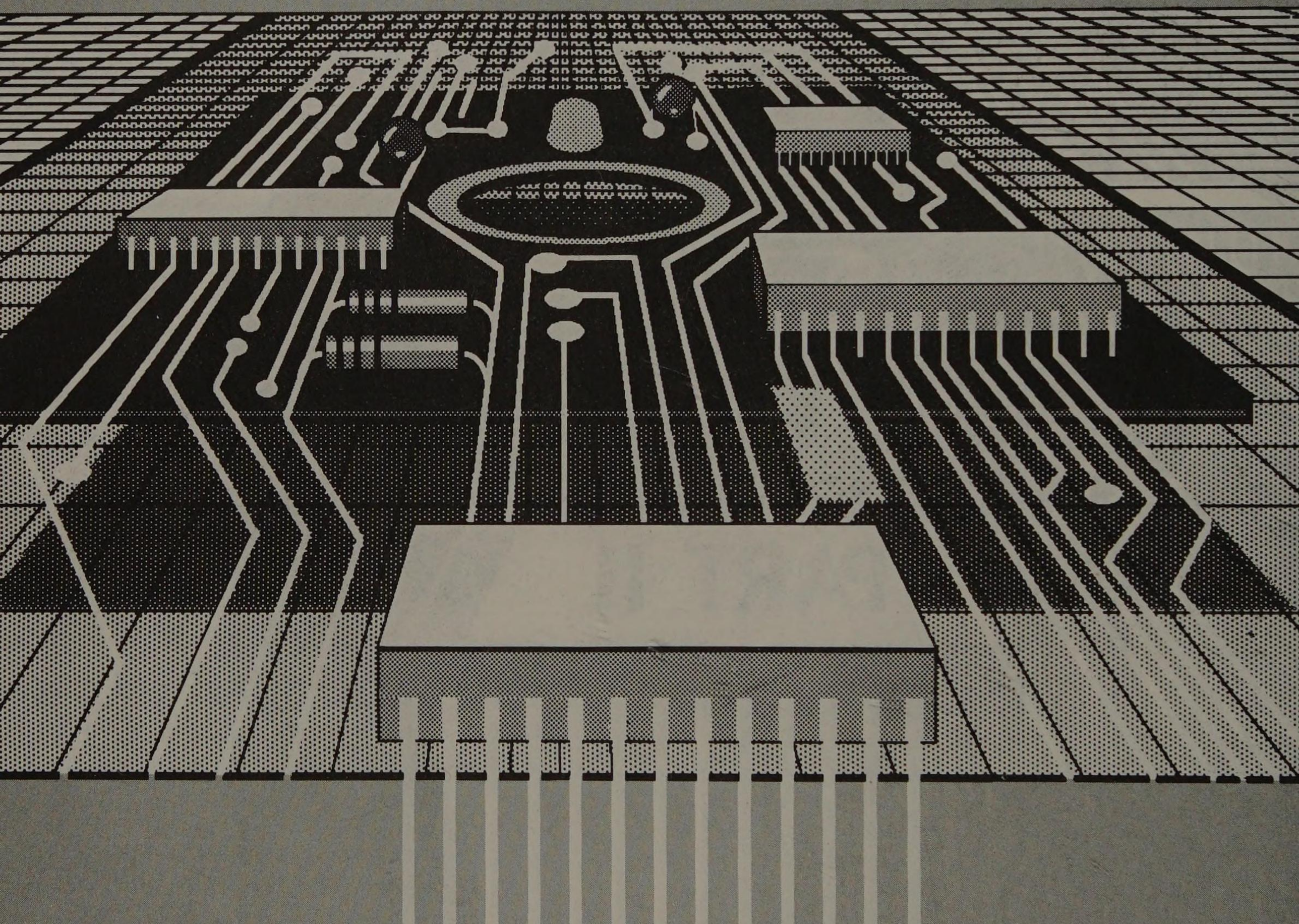
SUMMARY

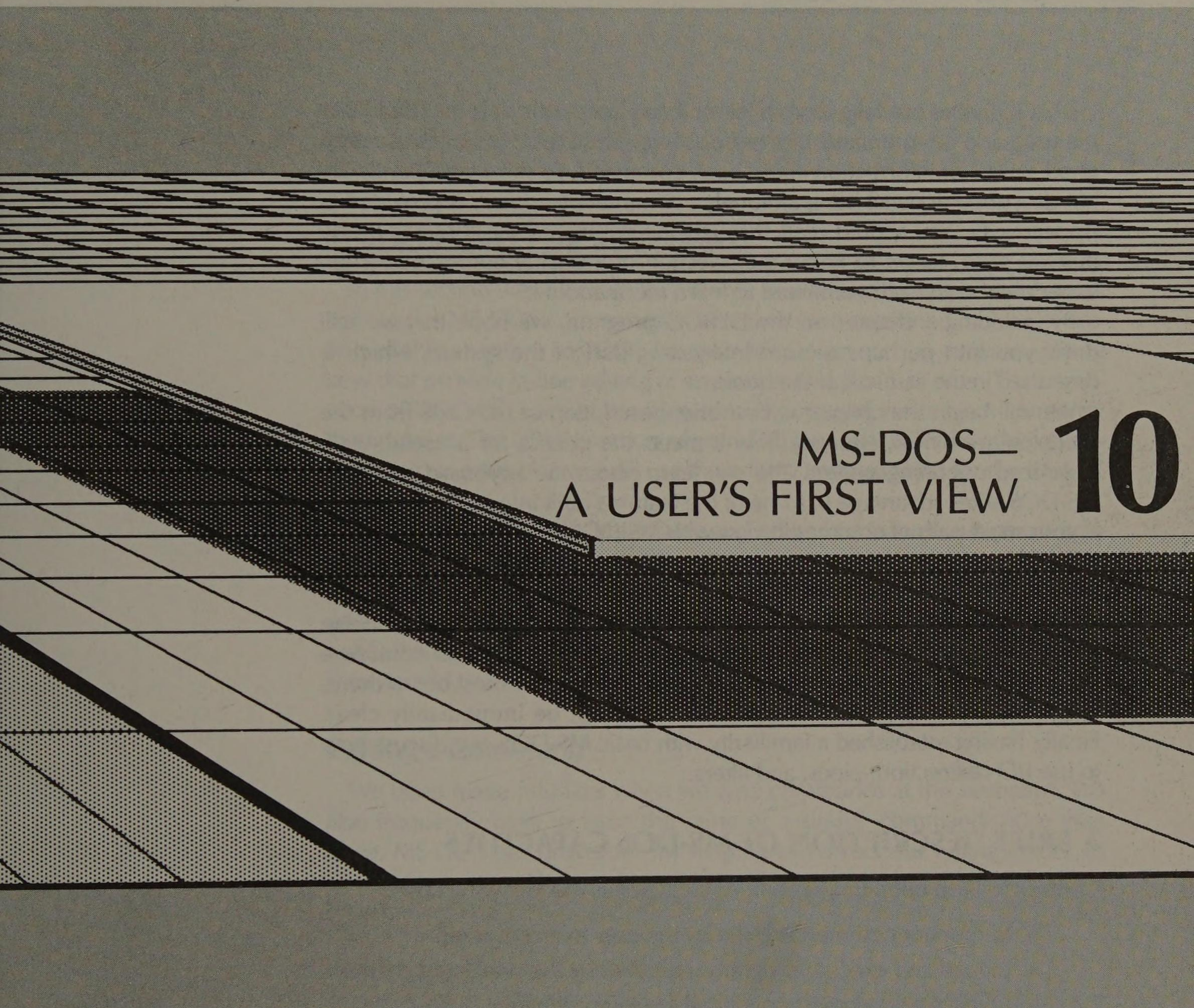
Network operations require the sharing of resources and individual files. Software running under network operations has to accommodate the needs of other programs that simultaneously attempt to access a single file, and users must be able to link with other resources on the net. To manage this, MS-DOS version 3 introduces the ability to link to other computers on the net, along with network access rights that control other's access to your shared resources, and file sharing modes and file locking controls that manage the sharing of individual files.

THE USER'S HANDBOOK



PART II





MS-DOS— A USER'S FIRST VIEW 10

INTRODUCTION

Welcome to MS-DOS. This chapter begins Part II, which is devoted to describing MS-DOS at the user's, instead of the programmer's, level.

User-level subjects that need to be covered in detail each have a chapter to themselves, after this one. This chapter is a place to discuss topics that do not require a whole chapter, but nonetheless need to be covered so that you can use MS-DOS efficiently and productively.

MS-DOS does not take long to learn. It has been written to be friendly to the user, and its commands are not obscure. The DOS manual is usually a good source of information about each command, so most commands are not described in detail here, although a command summary is provided in Appendix G. You should read through the command descriptions in the manual once, to get to know the selection of available commands. Then experiment with each command to learn more about it.

By including a chapter on the DEBUG program, we hope that we will draw you into perhaps a more interesting part of the system, which is described in the first half of the book.

We will begin the chapter with a brief description of MS-DOS from the user's point of view, showing how it meets the criteria for a useful, well-organized operating system. You will learn about the keyboard and how MS-DOS takes advantage of some of the features built into it. The keyboard is your main path of communication with MS-DOS; knowing how to use it effectively will save you time and effort. Next we will see what happens when MS-DOS encounters disk errors. After that, we point out parts of the DOS manual that have a high degree of information value relative to the amount you have to read. In the last part of the chapter, we will examine a few of the MS-DOS commands. We'll see how to get the most out of them, and how, on occasion, their operation may not be immediately clear. Finally, having established a familiarity with basic MS-DOS, we discuss how to use I/O redirection, pipes, and filters.

A BRIEF DESCRIPTION OF MS-DOS CAPACITIES

MS-DOS is an operating system. As such, it has the following tasks:

- to monitor commands given by the user to a computer,
- to load and execute programs specified by the user,
- to manage the devices attached to the computer,
- to manage the files of data and programs that the computer will work with, and
- to make available a small host of utility functions that provide the user with the tools and information he needs to be productive and efficient.

For example, an operating system needs to include convenient tools for:

- loading programs,
- transferring files from one disk to another,
- preparing a disk to accept files of data or programs, and
- sending information to a printer.

MS-DOS has all these features and more, including a convenient utility that keeps track of the time of day, and another that, given a date, computes the day of the week.

MS-DOS AND THE KEYBOARD

In this section we will examine features available at the keyboard that are not regular MS-DOS commands. (Commands are examined in the last section of this chapter.) MS-DOS has provided nine special-purpose editing keys that provide in-line editing of any command you may type. The association of keys with each of the nine special purposes varies from computer to computer. The EDLIN section of your DOS manual will show the keys to use. For our examples we will use the editing keys assigned for the IBM PC compatibles.

Before we examine in-line editing and other key features, we must note that some of these keyboard features are not available all of the time that you use your computer. In the section on each key feature, we will mention whether it is applicable over the whole scope of MS-DOS operations, or just at the command level (when you see the prompt). Let's see how to edit a command.

Editing a Command

We often make mistakes when we type commands at the keyboard. We also frequently have to issue the same or a similar command more than once. MS-DOS has provisions for helping us correct our typing errors, as well as for resurrecting the last command issued so we don't have to type it again.

Resurrecting the Last Command

The COPYALL (F3 [PCCs]) key does this for you. Unless MS-DOS has just finished executing a batch file, when you press the F3 key, you will see the last command reappear. All you have to do to execute it again is press the Return/Enter key. (Note: the names Return, Enter, and Return/Enter are used interchangeably for the line-termination key.)

You may want to change the last command a little before entering it again. The MS-DOS in-line editing features will meet just that need.

The In-Line Edit Feature

In the IBM PC compatibles, function keys F1 through F6 and the Esc, Ins, Del, and cursor-control keys can be used to edit a command. Until you press the Return/Enter key, you may rearrange, respell, shorten, or lengthen whatever you type at the MS-DOS prompt level.

The use of the in-line editing keys is described in detail in Chapter 11, under the heading "The Edit Command."

The Availability of the Edit Feature

In-line editing is guaranteed to work only at the command level, and then only if no keyboard key reassignment is in effect. At the command level (when you see the A> prompt), MS-DOS controls the interpretation of the function keys. Once another program has been called, it can alter the effect of the function (and other) keys, and so block out some, or all, of the in-line edit capability. For example, BASIC assigns different meanings to the function keys (BASIC also allows you to redefine them once again if you wish); Prokey, Sidekick, and other resident keystroke-translation packages allow different meanings to be assigned to any of the keys.

EDLIN and DEBUG do not change the effect of the function keys, so they retain the in-line editing capability. It is worth your while to learn how to use this feature of MS-DOS.

Now that we have seen how to resurrect a command, and learned that until you press the Return/Enter key you can play around with the text of a command all you like, let's consider long lines.

Entry of Long Lines

A long line is one that runs off the edge of the screen. At the command level, if you keep typing until characters reach the right-hand edge of the screen, the extra characters will wrap around to the beginning of the next line. You can cause wrapping to occur before the edge of the screen by typing a linefeed (Ctrl-J, and Ctrl-Return/Enter on the PC compatibles). This helps make what you type more legible. It may seem to you that this feature is not going to change your experience of any one day from average to exquisite, and you are probably right. However, it has proven useful once or twice.

Unlike most word-processing programs, when you use the linefeed key to move to the next line, no space will magically appear at the boundary; you need to supply one if you press linefeed at the end of a word, so that the long command you are entering has the correct format. There are circumstances when characters entered after the right margin is reached do not wrap to the next line. See the section "Mode-Setting Sequences" in Chapter 5.

Generation of Hard Copy

MS-DOS computers support one or two keyboard features for generating printed output of what you type and what appears on the screen. The first is

the "print screen image" feature, and is available on all the PC compatibles. The second is the "copy everything to printer" feature. Please refer to your DOS manual for the key combinations that control these services. (The PRINT command, which also generates hard copy, is described later in this chapter.)

Print Screen Image

When you press the "print screen image" key(s), the image currently on the screen is sent to the printer. The keyboard interrupt routines keep alert for this key combination. When the combination is pressed, everything else stops and a copy of the whole screen is sent to the printer.

Copy Everything to Printer

This feature copies to the printer every line as it is completed on the screen. Keyboard input is copied to the printer, as are lines produced by the software. A key combination toggles on or off the echoing to the printer of lines appearing on the screen. When you first invoke this feature, each line you subsequently type that appears on the screen will be echoed to the printer. The characters are echoed when you press the Return/Enter key at the end of a line, not one by one. To stop this echoing to the printer, press the same key combination again.

The Availability of the Print Image and Copy Everything Features

These features operate at a very deep level of the operating system; they are always available at the command level, but other practicalities might intervene.

Few applications programs have any need to interpret the meanings of the key combinations used for these features in any other way, so one might expect them to be active all the time. For most applications, the Print Screen Image feature is always available.

However, the Copy Everything feature involves accessing the printer at the end of every typed line. On networked systems, where several users may share one printer the Copy Everything feature offers much potential for chaos and is usually disabled. Accessing the printer at the end of every typed line can also slow the operation of a program intolerably. In this situation, and where the printer is being used by the program currently running, either the program turns off the Copy Everything and/or Print Image features, or when you use them, the resulting printed page is a mess.

Stopping the Rushing Text

When a program puts out many lines of data in rapid succession, you need a way to stop the flow so you can read it before it rolls off the screen.

Ctrl-S (Ctrl-NumLock on the PC compatibles) will do this. Ctrl-S is not a toggle—after you have stopped the flow of text, you start it again by pressing a different (any) key.

Ctrl-NumLock is a two-handed operation. You can't drink a cup of coffee and review screens very easily using this feature. Ctrl-S is easier to use. Sometimes the program displaying all that data will respond to the xon/xoff commands or a similar protocol. A program that recognizes the xon/xoff (transmission on/transmission off) protocol will stop transmitting characters when the receiving device, in this case the console, sends a message (xoff) saying it is not ready for more characters. This means that Ctrl-S and Ctrl-Q often stop and restart the flow of data. Sometimes just the Ctrl-S combination will both stop and restart the flow. Experiment—you may free a hand!

Keyboard Key Reassignment

Key reassignment allows you to redefine the meaning of keystrokes and keystroke combinations. You can use this feature to make character strings appear at the touch of a key. The character strings can be whole commands or just words that you use frequently, such as *anthropomorphic* or *MS-DOS*. This key reassignment feature is most useful, but for some manufacturers' versions of MS-DOS, there are commercially available keystroke-translation programs that offer more flexibility. Prokey and Sidekick, for example, allow you to reassign keys from within the middle of an applications program.

If you choose to reassign the meaning of the keys assigned to the in-line editing features, you will lose the in-line edit capability. You might want to consider using combinations of function keys with either the Shift, Ctrl, or Alt key, rather than reassign the meaning of the function keys alone.

MS-DOS AND DISK ERRORS

When MS-DOS cannot successfully perform a disk access that a program has asked it to do, it prints a message. In the middle of whatever program is running, a deep-level disk-error handling routine displays the message

```
Abort, Retry, or Ignore?
```

A correct response is one of the three characters A, R, and I (lowercase or uppercase). No carriage return (Enter key) is required. Choose your response carefully—the error routine responds immediately to the first key pressed.

If you respond to this message by pressing the A key, the deep-level routine believes you wish to abort the currently running program, not just the

disk transfer. When you type A, you will suddenly be back at the command (MS-DOS prompt) level. The program that was running won't be any more. Any data it had accumulated will be lost, including, for example, the whole (unsaved) text file you were creating. So avoid responding with an A if you can.

Each program may respond differently to disk errors. MS-DOS won't lie to a program about disk errors, so if you select the Ignore response (I), it is up to the program that is running to decide what to do about the error.

WHAT'S WHERE IN THE DOS MANUAL

We recommend that you read the DOS manual and keep it handy, with this book, for reference. However, there are parts of the manual that you are more likely to need than others, and parts that give information with such a wide range of applicability that, if you don't read them with close attention, joys that you might otherwise experience while using MS-DOS will be lost.

Command Syntax

Most DOS manuals compress much useful information into sections entitled *Syntax Notation* or *Format Notation* or *DOS Command Parameters.* (Appendix G of this book provides a summary of the MS-DOS commands and their syntax.) Other useful sections are *Device Names* or *Reserved Device Names, Global Filename Characters* or *Wild-Card Characters.* Read these with care.

The section *Information Common to All Commands* is probably the richest in the manuals. Once the information in this section has become part of you, you will learn new commands with ease, and you will know how to get the most out of familiar commands.

Reserved Device Names

The *Reserved Device Names* section presents the device names that can be used in place of file names. This minor form of I/O redirection adds power to your commands. For example, the command

```
COPY FILE.EXT PRN
```

is a print utility that operates on files composed of lines of text. The command

```
COPY CON PRN
```

sends to the printer (one line at a time), whatever you type at the keyboard.

(Note: COPY stops operating when it sees an end-of-file character. To enter an end-of-file character at the keyboard, use Ctrl-Z.)

If you keep the list of reserved file names in mind, you will be able to make better use of the power that MS-DOS offers.

Wild-Card Characters

This section, also known as *Global Filename Characters,* shows how the wild-card characters (? and *) can be used to find files that you don't know how to spell exactly, as well as groups of files having the same filename but different extensions.

Some commands, such as COPY, will accept global file name characters and operate on all the files that match, instead of just a single file.

Command Descriptions

This is where you learn what a given command can do. You will probably often refer to this section of the manual. Don't be afraid to reread the description of even a common command. The last time you read it, you may not have had an immediate need for some of its less obvious applications.

If you have experience working with computers, you may be tempted to skim over these command descriptions. However, some of the commands are complex and have valuable features that are easily overlooked. If you take the time to read these descriptions all the way through before you rush off to do something with the commands, your work will be more productive and more entertaining.

Here are examples of features of common commands that casual users may have missed.

The DIR Command

The DIR command has a built-in Ctrl-S (stop the flow) feature—the /P option. The command

```
DIR B:/P
```

will display the names of the files on the disk in drive B, pausing every screenful to give you time to examine the list.

You probably know about the /W option, which shows only the names of files, five to a line. This option provides a compact display, making it easier to look over a directory for groups of files.

The COPY Command

COPY will concatenate files for you, as well as copy them from one place to another. COPY will also accept CON (or in some computers CON:) as

the input or output, allowing you to display a file's contents using

```
COPY FILE.EXT CON
```

or to create a file from the keyboard using

```
COPY CON: FILE.EXT
```

The DISKCOPY Command

On IBM PC compatibles, you can copy to an unformatted disk with this command. DISKCOPY formats the target disk to be identical to the format of the source disk. This has a sometimes confusing side effect: if you copy a single-sided disk to a previously formatted double-sided disk, after the DISKCOPY operation the target disk will be single-sided.

The name of this command is sometimes contracted to DSKCOPY.

The CHKDSK Command

The CHKDSK command does not check the surface of the whole disk. It will not find unreadable sectors, except in the directory and file allocation tables. What it does is to check that the chains of clusters in the file allocation table are intact. CHKDSK's report includes, in the last two lines, a report on the amount of memory installed in the computer.

Old versions of CHKDSK can kill! Version 1's CHKDSK command fixes problems it sees without asking you if changing the disk's contents is acceptable. Running version 1 CHKDSK on disks formatted by later versions can cause chaos, since later versions' disks look wrong to version 1's CHKDSK.

PRINT SPOOLING WITH THE PRINT COMMAND

MS-DOS will print files while you run other programs. This background printing function operates well, is easily managed, and doesn't put much of a drag on the foreground task that runs concurrently.

The PRINT command will accept a list of files to be printed, put them in a queue, and print them as a background task while other programs are being executed. Options within the PRINT command allow you to

- display the queue (PRINT without any arguments),
- add more files to the queue (PRINT *file name(s)*),
- delete files from the queue (PRINT *file name* /C), and
- terminate the whole printing-from-queue operation (PRINT /T).

The queue can hold a maximum of ten files at one time. When you add files to the queue, they have to come from the current directory, but you can change to another directory after adding files and then add more files to the queue.

Obviously, you cannot use the printer for other purposes while the PRINT function controls the printer. Also, the first time the PRINT command is called, about 3600 bytes are added to resident MS-DOS.

GRAPHICS TO THE PRINTER

This is a nice feature to have. It prints on a dot-matrix printer an image of the screen, sideways if necessary. (In the IBM PC compatibles, it is useful when MS-DOS is operating with a color/graphics adaptor. It has no relevance when MS-DOS is operating with the IBM monochrome adaptor.)

Invoking the GRAPHICS command has two effects: (1) resident MS-DOS is increased by about 700 bytes, and (2) when operating in any graphics mode, the print-screen-image keystroke prints the image on the screen driven by the graphics adaptor.

When the number of pixels on the screen is greater than can be printed across one printer line (for example, when the color/graphics adaptor is in 640 × 200 mode) the image is printed sideways, rotated 90 degrees counter-clockwise. The bottom of the screen appears on the right-hand side of the paper.

PIPES, FILTERS, AND I/O REDIRECTION

These features expand the power of MS-DOS. As further versions of MS-DOS are released, these features will become a more natural part of the system. We look at I/O redirection first.

I/O Redirection

We need to know the meaning of standard input and standard output to be able to understand I/O redirection.

The standard input to a program is from the keyboard, the standard output is to the screen. A program can accept input from sources other than the keyboard, and it can output data to devices other than the screen, but by convention, the standard input device is the keyboard and the standard output device is the screen.

I/O redirection allows you to redirect a program's output, which normally goes to the standard output device, so that instead of going to the screen, it will go to the device or file you specify. The following statement redirects the output from the DIR command:

```
DIR A: >B:DIRFILE
```

The directory that normally shows up on the screen when DIR is run will be entered, line by line, into the newly created file (DIRFILE) on drive B.

Redirection also allows you to change the source of program input from the keyboard to another device or file that you specify. The following statement tells a program UPPER (which converts all characters it sees to uppercase) to accept its input from the file SPARE.BAT instead of the keyboard:

```
UPPER <SPARE.BAT
```

A variant of standard output redirection is to *append* the standard output from a command (program) to an existing file. The following command will add the directory of drive B to the text of the directory already in DIRFILE (DIRFILE was created by the first example command):

```
DIR B: >>B:DIRFILE
```

The symbol sequence >> means "append the standard output to an existing file." The > symbol, if used with a file name, means "put the standard output into that file, starting at the beginning of the file."

MS-DOS's I/O redirection affects only output that normally goes to the screen and input that normally comes from the keyboard. That is, MS-DOS can only redirect data that is passed through its own screen and keyboard I/O functions.

Pipes

Imagine data flowing through a pipe. At intervals within the pipe are programs that process the data. The data that comes from a program upstream flows into the program downstream, and so on, until the pipe ends.

MS-DOS allows you to build pipes. Pipes are basically strings of programs where the screen output (standard output) of one program becomes the keyboard input (standard input) for the next program. Here is an example of a short pipe, where the directory information becomes the input to a sorting program:

```
DIR ¦ SORT
```

The ¦ symbol means "take the standard output of the command that precedes the ¦, and provide it as standard input to the command that follows."

Here is another example. Create a file that contains only a single carriage return by typing

```
COPY CON CRLF.TXT
(type a carriage return on this line)
Ctrl-Z
```

and add the line

```
TYPE CRLF.TXT ¦ PRINT
```

to the AUTOEXEC.BAT file. When you boot the system the print command will be loaded and the default printer selected. This is a simple way to eliminate the need for human intervention when PRINT is first invoked.

Pipes can be longer than the examples shown. When you have a library of programs that each do just one thing well, you can use pipes to send data through several of those single-function programs in a row.

The data that is in a pipe is kept by MS-DOS in files called %PIPE*nnn*. You may see these files in the root directory from time to time. The *nnn* part of the filename is used by MS-DOS to keep track of which part of a pipe the %PIPE*nnn* file represents.

Let's look at filters, and then see how data can be put through several filters in a pipe before ending up in a file for later reference.

Filters

Filters are programs that are written to do one thing well. They accept data from the standard input device, process it efficiently, then send it to the standard output device.

MS-DOS comes with three filters: the programs FIND, SORT, and MORE. These are all text-oriented filters. They accept text, work on it in some way, and then output it. Here are some examples of other text-oriented filters:

- A program that puts each word of a text file on a separate line
- A program that throws away any line in a file that is a repeat of a previous line
- A program that compares text files line by line, creating files for lines seen in one file but not in the other

These examples have not been chosen at random. They lead to the classical example of the use of filters.

If you have a dictionary on disk, one word to a line, sorted in alphabetical order, you can use the SORT filter and the three example filters to create a spelling checker pipe. The pipe looks like this:

```
WDPERLINE <THESIS ¦ SORT ¦ DUMPRPTS ¦ DIFF DICT>
SPELLING.ERS
```

Your THESIS file becomes the input to the WDPERLINE filter that puts every word on a single line. The output of the WDPERLINE program becomes the input to the SORT program, which puts all the lines (words) in alphabetical order. The DUMPRPTS filter sees the output of the SORT program, and only passes on to the DIFF program a single copy of each word. DIFF compares the dictionary file DICT with the output of DUMPRPTS, and puts any words in output from DUMPRPTS that are not in the dictionary file into the file SPELLING.ERS. So your pipe is a spelling checker.

Note that this example uses I/O redirection, pipes, and filters. From it, you can see why the characters <, >, and ¦ are not permitted in file names.

The spelling checker pipe could become the first line of a batch file SPELLING.BAT. THESIS would be replaced by %1, so the pipe could work on any text file. The second line of the batch file might be

```
PRINT SPELLING.ERS
```

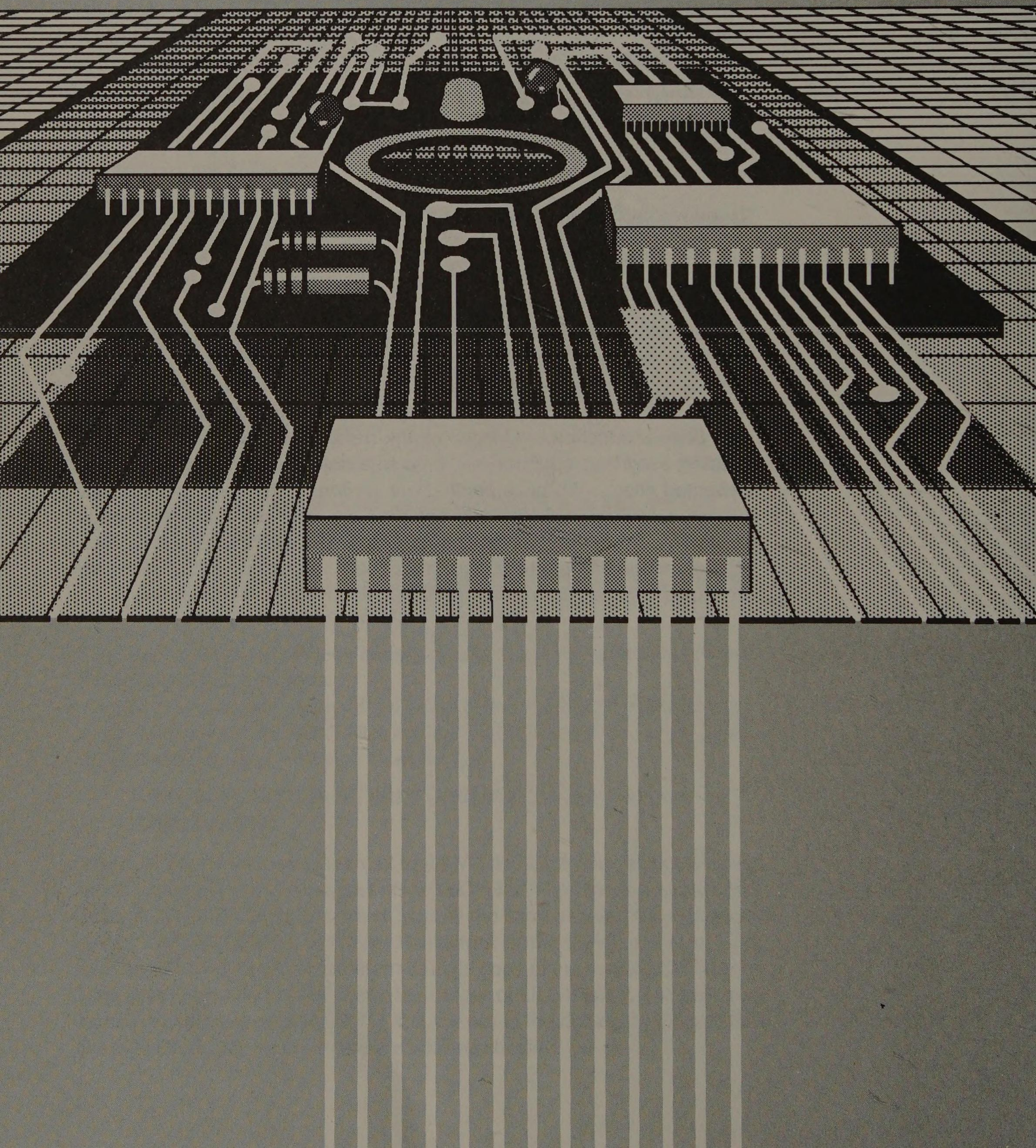
and the third line might be

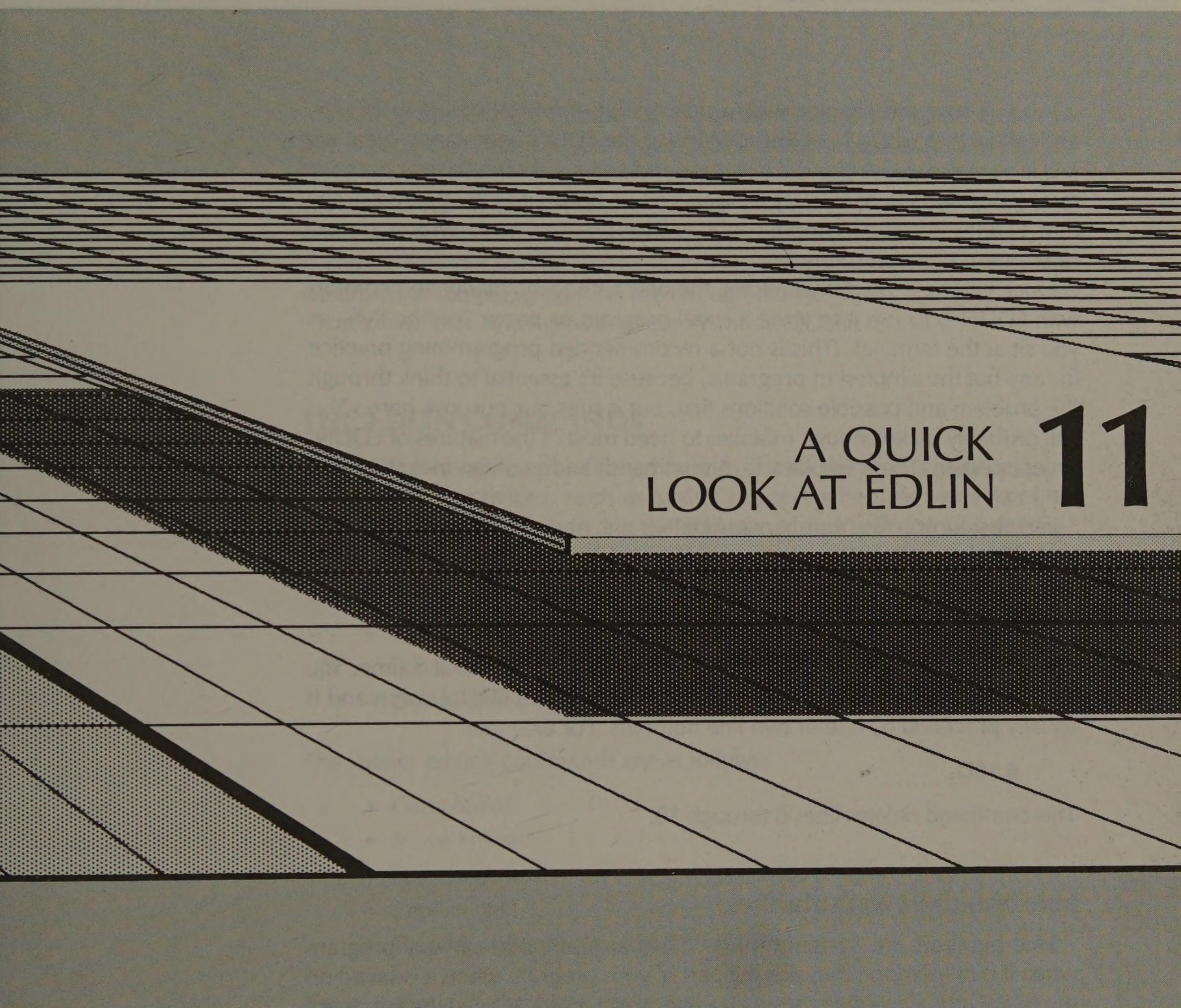
```
TYPE SPELLING.ERS ¦ MORE
```

so you would see a list of spelling errors, displayed one page at a time.

SUMMARY

In this chapter, we have briefly examined the purpose of an operating system, and we have noted that MS-DOS is a user-friendly system that provides us with a powerful set of useful commands. We have reviewed some of the MS-DOS features that are available at the keyboard: in-line editing, hard copy production, and stopping the rushing text. We have seen how selecting the Abort option in response to a disk error may have dramatic unwanted effects. We have pointed out sections of the DOS manual that are particularly valuable to study, and we have reviewed a few commands to show you some of their less obvious features. We have also seen how I/O redirection and pipes help us make better use of MS-DOS. Let's move on to the next chapter to examine EDLIN, a tool you will be using often.





11 A QUICK LOOK AT EDLIN

INTRODUCTION

EDLIN is a funny little editor, but it's free. You will probably be using it, so it merits our discussion. This chapter is an outline of EDLIN's capabilities and limitations.

EDLIN is a line editor. It can understand simple one-character commands; each command is usually preceded by a line number or a pair of line numbers indicating the line or lines of the file on which EDLIN should operate.

We will start the chapter with a description of line numbers in EDLIN, and follow that with a brief introduction to the EDLIN commands. Next we will look at the steps required to create and edit a new file. In the course of this guided tour, all of EDLIN's features will be brought to light; you will see that EDLIN is not a complex editor. The chapter ends with a quick-reference command summary.

Once you have read this chapter, the best path to familiarity and comfort with EDLIN is to use it to write a small program or poem "on the fly"—as you sit at the terminal. (This is not a recommended programming practice for any but the simplest of programs, because it's essential to think through the problem and possible solutions first, but it suits our purpose here.) You will probably make enough mistakes to need most of the features of EDLIN, except perhaps the R (replace text) command, and you can invent a need for that.

Let's first look at line numbers in EDLIN.

LINE NUMBERS IN EDLIN

EDLIN is a line editor; it is designed to work on one line at a time. You have to specify line numbers to get anything done. An EDLIN command is usually preceded by one or two line numbers. For example:

```
6,10D
```

This command deletes lines 6 through 10.

Line Numbers as Butterflies

Line numbers are transient things. They are only part of your program when it is in memory; they are not part of your program when it is saved on the disk. During an editing session, a line number's location in the text is not necessarily fixed. One line number may flit from line to line as the session progresses. This is because EDLIN renumbers lines immediately after every insertion or deletion (and after every Append or Write command).

If you want to make several changes to a file, you may often find yourself using the List and Page commands. With these commands you can keep track of what number belongs to which line at each stage of the edit. After you have gained some proficiency in using EDLIN, you may be able to recompute line numbers in your head after each command.

(Note: To minimize changes in line numbers, make your editing changes first at the end of the file, working backwards; that way, the early line numbers will remain unaltered until you need them.)

EDLIN maintains the number of the current line; it will use the current line in any command if you don't give it a line number to work with. EDLIN's choice of the line number to make current is not always what you would expect. EDLIN's interpretation of current line number is like the many odd things one gets used to and eventually finds useful.

EDLIN provides several shortcuts for expressing line numbers, which are described in the paragraphs on the List command.

Let's now look at the groups of commands in EDLIN.

THE EDLIN COMMANDS

The EDLIN commands will modify text, display the results, and perform housekeeping activities, such as saving the text when the work is completed. For a table summarizing the commands and their formats, see Figure 11.2 at the end of the chapter. These are the simple text-modification commands:

- Insert line(s)
- Edit a single line
- Delete line(s)

The fancier editing commands are as follows:

- Copy line(s)
- Move line(s)
- Search for text
- Replace text
- Transfer a disk file into the current file

These commands show you the text:

- List line(s)
- Page through the text

The last group consists of the housekeeping commands:

- End or Exit EDLIN, saving your work
- Quit without saving
- Append more lines from the rest of the file on the disk
- Write some lines to the disk to make room for more

EDLIN takes the first letter of each of these commands as its cue for action,

with the exception of the command to edit a single line, which takes a line number only.

Command Format

The format for EDLIN's editing and display commands is

line number (, *line number*) *command*

where both line-number values are optional. The Copy and Move commands have additional arguments. The Append and Write commands take a quantity of lines as an argument. The Transfer command includes a file name as an argument. The End and Quit commands take no arguments.

The command E means "write the file to disk and quit to MS-DOS." To edit a line, just type the line number.

Multiple Commands per Line

You can string several EDLIN commands together on a single line. The commands do not need separators between them (except after the Edit command); EDLIN can figure out what you mean when it sees the command letter. The Edit command, which is just the line number (no letter), must be followed by a semicolon. This prevents it from being confused with an optional line-number argument to the next command.

You will find examples of the commands in the descriptions that follow. Now that you have seen the available commands and know about line numbers, let's look at the steps necessary to create a file.

CREATING A FILE USING EDLIN

If you have used line editors before and want a little practice with this one, you could type along inventively as you read this section.

To create a file, you first need a file name and a disk on which to store it. These are necessary because before EDLIN will let you enter any text, it creates a temporary file, called MYFILE.$$$, on the disk (where MYFILE is the filename for your file). Note that the temporary file's extension is always .$$$, regardless of the extension (if any) you choose for your file.

Let's assume you have named your file MYFILE and you want it to reside on the disk in drive B. To invoke the editor and create the file, type

```
EDLIN B:MYFILE
```

At the asterisk (*) prompt, use I (Insert) to get going, then enter your text until you are finished or until you make a mistake that you wish to correct immediately. Use Ctrl-C (the Break character) to leave the Insert mode.

Now that you have your file, albeit not yet correct, let's see how to use EDLIN's List and Page commands to look at it. After that, you might wish to experiment with the Insert and Edit commands to make a few changes.

THE LIST COMMAND

The List command is the letter L. Typed without any arguments, this command lists a screenful of lines (if there are that many) centered around the current line. If you are typing along with this example, type L and see the result.

Specifying a Range of Lines

To specify the lines you wish to see, use *n,m*L. This command will list line numbers *n* through *m* (or less if *m* is beyond the last line in memory).

Shortcuts You Can Use when Specifying Line Numbers

The pound-sign character, #, can be used to indicate the line number one greater than the last line in memory. For example, *n*,#L will list from line *n* to the end of text in memory. (Note: See the Append and Write command descriptions for explanations of why the last line in memory is used here instead of the last line in the file.)

When you list your text, the asterisk appears to the right of the line number of the current line. To save you some of the trouble of remembering line numbers, EDLIN will accept a period to denote the current line in case you can't remember (or find) it.

Relative Line Numbers

EDLIN understands line numbers relative to the current line, as well as absolute line numbers, the # sign, and the period. Examples of relative line numbers are −2 and +6. The command −2I means "insert text prior to two lines before the current line." The command −5,+10L means "display from five lines before to ten lines after the current line."

You now have the full power of the List command at your fingertips.

THE PAGE COMMAND

The Page command gives you a single-keystroke command for paging through your file quickly. If you don't specify any line numbers, each Page command will display the next screenful of lines in your file.

The Page command differs from the List command in two ways: (1) it adjusts the current line number to be the last line displayed each time the command is used, and (2) the default value for the first argument is the current line plus one, unlike the List command's default value of current line minus eleven. These differences explain why the Page command can page through your file for you if you use the default arguments. If you supply both arguments, Page works like the List command.

If you have been typing along here, you have now seen the text you entered. If not, at least you now know how to look at text, so let's go on to look at the Insert command. Then we'll see what EDLIN offers for making changes to text that already exists (i.e., the Edit command).

THE INSERT COMMAND

The Insert command inserts additional lines into your text. It will not insert extra characters within a line; see the Edit command for that. (The Insert command is also the first command you use on an empty file.)

Imagine a two-line program that for some reason you find deficient. The program needs additional code between lines 1 and 2. Use the command 2I. The lines that you type will be inserted *before* line 2, and will be numbered starting at 2. Your original line 2 will be renumbered.

When you have finished inserting text, you will want to have the other EDLIN commands available again. To exit the Insert mode, use Ctrl-C (the Break character) on a new line.

Argumentless Inserts

If you do not specify an argument for the Insert command, text will be added before the current line. If you want to add text after the last line in memory, use #I or a very large line number. EDLIN won't allow gaps in its line numbers, so very large numbers are interpreted as if you had specified a line number one greater than the last line.

THE EDIT COMMAND

If you have been typing along with this guide to text creation, you will probably need to use the Edit command (remember, not E—just specify the line number) to alter the contents of a line or two.

You may already be familiar with these line-editing conventions; the EDLIN editing commands are consistent with the use of the MS-DOS editing keys, which are described in your DOS manual. Extensive but not very

gripping examples of these commands can usually be found on the succeeding pages of that DOS manual.

EDLIN's in-line editing keys are COPYONE, COPYTO, COPYALL, SKIPONE, SKIPTO, ESCAPE, INSERT, and NEWLINE, along with cursor left, cursor right, and backspace. These keys operate as edit keys by controlling the reproduction of a line that already exists. For example, the COPYONE key will copy one character from the existing line onto a new line. The new line will replace the old one when editing is complete. Figure 11.1 shows the correspondence between the functions and the keys to which they have been assigned in some of the different MS-DOS computers.

The functions of the keys are described below, but it is *not* just as easy to learn about them by concept as it is by experience. If you have never used these capabilities, it is worth taking the time to practice a little. Here are the in-line editing keys and their effects (the key assignments for the IBM PC compatibles are shown in parentheses):

- COPYONE (F1) copies a single character onto the new line.
- COPYALL (F3) copies the rest of the characters onto the new line.
- COPYTO (F2) copies all characters up to a specified character.
- SKIPTO (F4) skips all characters up to a specified character.
- SKIPONE (Del) skips over one character when copying characters onto the new line.
- INSERT (Ins) allows insertion of new characters.
- Cursor right usually has the same effect as the COPYONE key.
- Cursor left and backspace undo the effect of the COPYONE key.

	IBM PC	Compaq	Z-100	TI Professional
COPYONE	F1	F1	F1	→
COPYTO	F2	F2	F2	F2
COPYALL	F3	F3	F3	F3
SKIPONE	Del	Del	F4	Del
SKIPTO	F4	F4	F5	F4
ESCAPE	Esc	Esc	Shft F0	↓
INSERT	Ins	Ins	F5	INS
NEWLINE	F5	F5	F0	F5

Figure 11.1: *Keys Assigned to In-Line Editing Features*

- NEWLINE (F5) is useful when you are content with how the line looks now, but don't want to hit Return/Enter yet. NEWLINE replaces the old line with the new and allows further editing.
- ESCAPE (Esc) allows you to abandon the line and start over.

For editing short lines, the most useful keys are COPYONE, COPYALL, SKIPTO, SKIPONE, and INSERT. To change one character in a line (perhaps to correct a mistyped command), press COPYTO (F2), then the character you wish to change, then the replacement character, and then COPYALL (F3). In lines where the character you wish to change appears in several places, and you only wish to change one occurrence of it, you can hold down COPYONE (F1) or the cursor-right key until you reach the character just before the one you wish to change. Then type the character of your choice, and press COPYALL (F3) to finish the line.

The INSERT (Ins) key toggles the Insert mode within a line. As you begin editing each new line, the Insert mode will be off. It needs to be toggled on if you wish to add more characters to the line. For example, to change SEGMNT to SEGMENT in a line, select the line, press COPYTO (F2), then N (or COPYONE (F1) until SEGM appears), press INSERT (Ins), type E, and press COPYALL (F3).

The SKIPONE (Del) key is used to say that you don't want the next character that would show up if you pressed COPYONE.

COPYTO (F2) and SKIPTO (F4) are the "search for" keys. They are less useful for long lines than for short ones; they lose their usefulness when there are multiple appearances of characters within a line. For instance, in the above insertion example, you might find yourself inserting E's in places you don't want if there is more than one N in the line.

To learn the use of these keys, I recommend trial and error, followed by any enlightenment that the DOS manuals may have to offer. I also recommend that you find a decent full-screen editor and not tolerate the limitations of any version of EDLIN.

Now that you have seen the action of the editing keys within a single line, we will look at the other commands EDLIN provides.

THE SEARCH AND REPLACE COMMANDS

The Search and Replace commands can both be used to find text strings. They both have a question-mark option that allows you to make a choice of action at each occurrence of the string found, and both allow you to specify a range of lines. The question mark, if desired, is inserted between the line numbers (if any) and the command. When the question-mark option is used, the Search command will ask if you want to continue the search for

others after one matching string has been found. When the question mark is used in the Replace command, you have the choice of replacing each matching string as it is found. The Replace command's search for a matching string continues until the end of the file or until you type Ctrl-C (the Break character).

The format of the Search command is

(*n*)(,*m*)(?)S*string*

The format of the Replace command is

(*n*)(,*m*)(?)R*string*1<Ctrl Z>*string2*

The items in parentheses are optional. In both the Search and Replace commands, the string searched for follows the command letter. If you wish to replace the found text, the string searched for is terminated by Ctrl-Z (F6) and followed by the replacement text.

The Search and Replace commands allow you to step through the file quickly. The default value for *n*, the first argument, is the current line plus one, instead of line one. This means that you don't have to specify line numbers to step incrementally through the file. Second, the commands remember the search string and the replace string that were last used, so you don't have to retype them.

The Search and Replace commands are powerful components of EDLIN. We now look at two housekeeping commands that are used to edit files larger than EDLIN's estimate of available memory.

THE APPEND AND WRITE COMMANDS

The Append and Write commands are used in working on large files. EDLIN considers a file large when it occupies more than about 75 percent of the available memory. EDLIN needs the other 25 percent for work space. When a large file is being edited, not all of it will be in memory at once; some of it, either the beginning, the end, or both, will be on disk. EDLIN's editing window moves sequentially across the file from beginning to end.

The Append command gets more lines to work on. They come from the unedited part of the file that still resides on disk. The Write command moves a number of lines to the disk, making room for more lines in memory. The format for these commands is simple. It is the count of the number of lines to transfer, followed by the single character A or W.

THE TRANSFER COMMAND

The Transfer command could just as well be called Merge files or Read a file, but M and R are already used by the Move and Replace commands.

The Transfer command will add to the file being edited, before the current (or specified) line, the *entire* contents of the file named in the command. Thus, Transfer allows you to merge several files into one, and to concatenate files. Standard pieces of text or code can be kept in individual files, and then included in your programs as you need them by using the Transfer command.

The Transfer command has one characteristic that makes it fit right into the EDLIN family—it works, but its reach is short. The Transfer command does not understand path names, just a file name, which can be preceded by a drive identifier. Transfer can reach as many directories as there are disk (including RAM disk) drives. To make full use of the Transfer command, you need to have your source files well organized before you begin your editing sessions. The Transfer command redeems itself in a small way by accepting file names (i.e., a filename plus an extension).

The format of the Transfer command is

*(n)*T*(d:)filename(.extension)*

THE COPY COMMAND

This command copies blocks of lines from one place to another in the file being edited. It will produce multiple copies of a single block of lines if so requested. The format of the Copy command is

*(n),(m),dest(,count)*C

The items in parentheses are optional.

Lines *n* through *m* are copied to the place immediately before the line specified as the destination (*dest*). If the *count* field is included in the command, that number of copies of the block of lines is created before the destination line.

To move the current line to appear before line 10, type

```
,,10C
```

This has the same effect as typing

```
.,.,10C
```

Note, however, that it is not possible to use the Copy command to create interesting interleaved patterns by overlapping the range of lines and the destination line.

After the command is complete, the current line is the first line of all the copies that were made.

Let's look at a similar command, where blocks of lines are moved instead of copied.

THE MOVE COMMAND

The Move command moves groups of lines around in your file. The lines are moved to the place *before* the specified line, as in other EDLIN commands. The format of the command is

(*n*),(*m*),*dest*M

The items in parentheses are optional and default to the current line. The command to move the current line to appear before line 10 is

,,10M

The destination line must not be included in the range of lines to be moved.

EDITING BINARY FILES

EDLIN usually works on ASCII files, but you can use the /B option after the file name to tell EDLIN that the file being edited is in binary format. The difference between binary and ASCII files is that some of the bytes in the file are interpreted differently by the software that processes them. In a binary file, you can have any of the 256 possible characters that can be made from eight bits. In an ASCII file, control characters (the ones that are permitted) are interpreted as commands by the software that processes the file and often do not show up in the text that we read. The key example of this is the end-of-file character, Ctrl-Z (1AH). That character can occur in a binary file long before the end of the file. (Note: MS-DOS maintains a file-length indicator in the directory. That is where programs can find out how long a file is if they don't want to rely on spotting the Ctrl-Z as they process it.)

Now that you know binary files can be edited, you may be wondering how to get the control characters into the file, past all the software that usually screens them out. The answer is Ctrl-V. If EDLIN sees a Ctrl-V, it will wait for the next character that you type (it must be uppercase) and make that the control character. For instance, entering Ctrl-V followed by Z will tell EDLIN that the character to enter or search for is Ctrl-Z. EDLIN will accept some control characters even if you don't use the Ctrl-V method to enter them, but that approach is not recommended.

We've reviewed all of EDLIN's commands, so it might be time to leave the editor. Let's see how to do that.

ENDING AN EDITING SESSION

When you have finished making changes, type E to exit. EDLIN will save your text on disk. If you want to abort a practice session, type Q. Your changes will not be saved.

EDLIN'S TREATMENT OF .$$$ AND .BAK FILES

It might be useful to know what EDLIN does with .$$$ and .BAK files. In the unlikely event of your system crashing during an edit, the information here will help you reconstruct your file from the wreckage, or, more likely, lead you quickly to the conclusion that the work done during the crashed session is lost. This section is also relevant if you edit files that have the same filename but different extensions. Let's look at that case first.

EDLIN deletes .BAK files. It also creates them, but it deletes them first. What this means is that if you have a .BAK file with the same filename as the file you are about to edit, and you want to keep that .BAK file unchanged, you need to change its name before you begin the edit. It also means that you cannot successfully call upon EDLIN to edit any file with the extension .BAK. I will explain.

You know that an MS-DOS file name consists of an eight-character filename, optionally followed by a period and a three-character filename extension. When EDLIN is operating, it takes over the use of three file names:

1. the file name of the file entered at the time EDLIN was called;
2. the filename part (file name without extension) of the entered file name, with the extension .BAK; and
3. the same filename with the extension .$$$.

When you first call EDLIN, *filename*.$$$ is created. The disk space occupied by the .BAK file is not deleted until either the End edit command is given or the disk space occupied by the .BAK file is needed by the .$$$ file. *Filename*.$$$ is where the editing takes place, although its disk space isn't used until either available memory is about 75 percent full or the End edit command is given.

At the end of the editing session, the file whose name you specified in the EDLIN call, if it was there, is renamed *filename*.BAK (it loses the distinguishing extension, if it had one). *Filename*.$$$ is given the file name you specified when calling EDLIN. By this process of deleting and renaming files, the user ends up with the usual two files at the end of an editing session: one named *file name* and the other named *filename.BAK*.

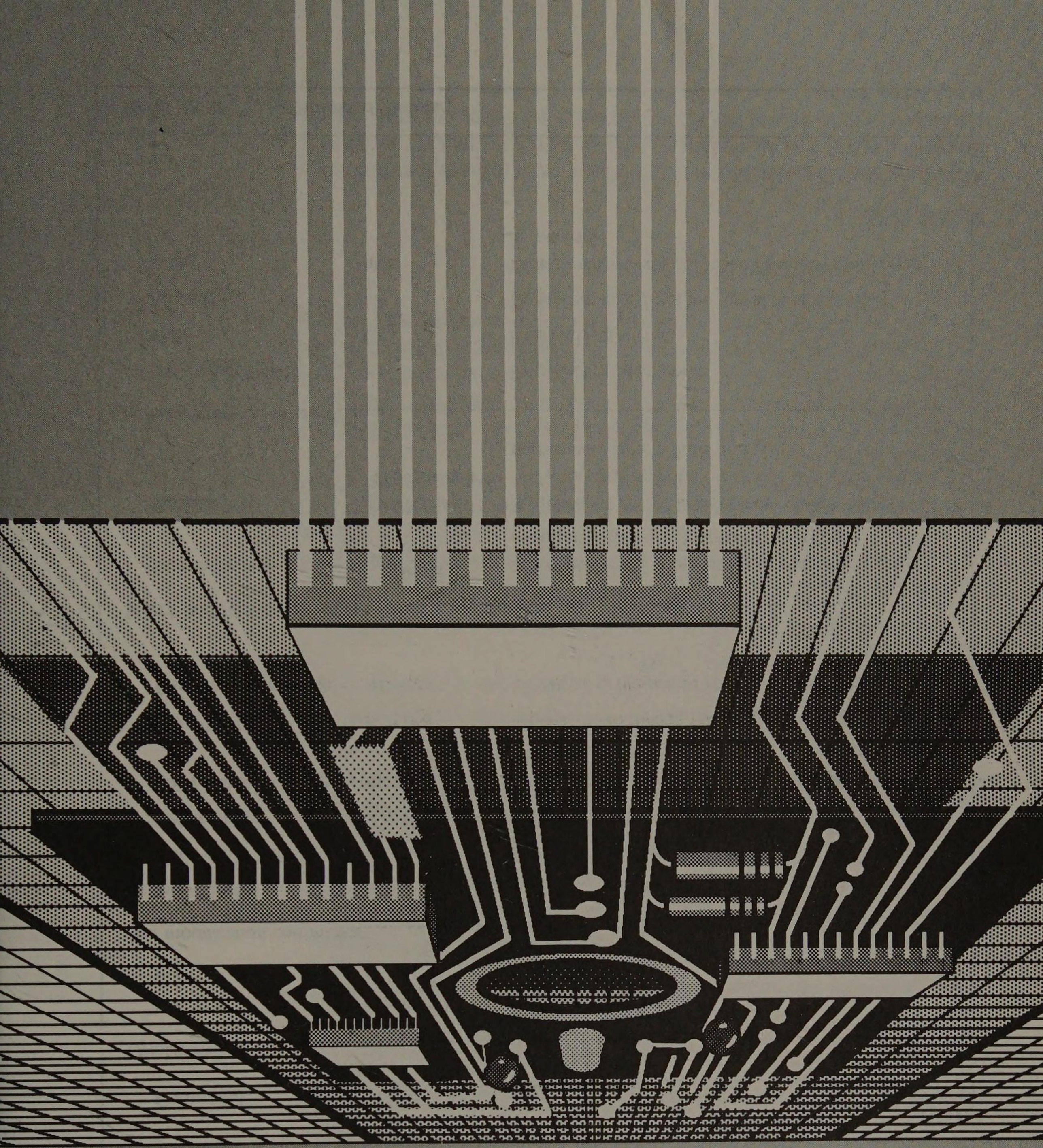
Figure 11.2 is a summary of the EDLIN commands.

SUMMARY

We have seen that EDLIN contains the basic set of features that make up a simple line editor. It is adequate and reliable, and it is useful for quick edits because it is built into MS-DOS. For luxury, high living, and productivity, buy a better editor.

Command	Format	Explanation
Display commands		
List	*n,m*L	Lists lines *n* through *m*
Page	*n,m*P	Pages through the file
Text modification commands		
Insert line(s)	*n*L	Inserts line(s) before line *n*
Edit a single line	*n*	Line number only
Delete line(s)	*n,m*D	Deletes lines *n* through *m*
Copy line(s)	*n,m,d,c*C	Copies lines *n* through *m* to appear before line *d*, *c* times
Move line(s)	*n,m,d*M	Moves lines *n* through *m* to appear before line *d*
Transfer (merge) a file	*n*T*filespec*	Copies all of *filespec* to appear before line *n*
Search and replace commands		
Search for text	*n,m*?S*string*	Searches for *string* in lines *n* through *m*. If a ? is included, pauses each time it finds *string*.
Replace text	*n,m*?R*string1* <F6>*string2*	Searches for *string1* and replaces it with *string2*. '<F6>*string2*' may be omitted. If a ? is included, pauses each time it finds *string1*.
Housekeeping commands		
End or Exit EDLIN	E	Exits, saving your work
Quit	Q	Exits without saving
Append	*n*A	Appends *n* lines from the rest of the file on disk
Write	*n*W	Writes *n* lines to disk to make room for more lines in memory

Figure 11.2: *EDLIN Command Summary*



12 YOUR MS-DOS FILES

INTRODUCTION

This chapter is all about the file system offered by MS-DOS. You will see how the file system is organized, the components of that organization, and how to use the tools that manage those components. By the end of the chapter you will have learned enough about the MS-DOS file system to develop easily manageable file structures.

We will start by examining a tree-structured set of files. Tree-structuring is the type of file organization supported by MS-DOS. You will see how each disk drive contains a root directory of files, and how all other files on that disk are addressed from that root directory. After that, we look at file

capacities and the placement of files on a disk, and how those might affect the organization of files and the performance of your file system. We end the chapter with a discussion of file names, an area where the power of your imagination can either damn you to confusion or provide graceful support in your life with computers.

FILES IN MS-DOS

A file is simply an organized collection of bytes (characters) that the disk operating system knows how to find, copy from, and make changes in. The files exist on a disk. The disk contains directories of the files to be found on the disk. The prime directory is found in a predefined location at the beginning of the disk and is called the *root directory.*

A directory contains information on the size of the file, the date and time the contents were last changed, and the location on the disk of the first block, or "cluster," of the file. (Note: The term *cluster* will be explained a little later. For now, we will use *cluster* and *block* interchangeably.) A file grows and shrinks as the data in it expands and contracts. MS-DOS maintains lists (more accurately, chains) of the blocks that belong to each file; these lists, which are found in the file allocation table, grow longer and shorter as the file grows and shrinks.

The maintenance of the directories and file allocation table and the insertion and extraction of data take place at a level invisible to the user, except that assorted whirring and clicking occurs as the files are updated. For more details on how the operating system accesses files and directories, refer to Chapters 2 and 3, which talk about programming for file creation and access and about the disk layout in MS-DOS.

An Aside on Disk Volumes

MS-DOS identifies disks by drive letters. Typically, the first hard disk is drive C:, the next D:, etc. Strictly, the drive letter identifies a *disk volume,* not a physical disk drive. MS-DOS was built to handle a maximum disk size of 32 megabytes. Physical disk drives whose capacity is greater than that must be partitioned into disk volumes, each with their own drive letter and root directory. Smaller physical disks can also be partitioned into disk volumes.

Tree-Structured Files

MS-DOS uses a tree-structured, or hierarchical, filing system. A tree-structured filing system can hold many directories, where each directory is

applicable to a set of files that can be grouped conveniently together. This ability to organize files into independently accessible groups gives much flexibility in the management of files in MS-DOS.

This section will show you how MS-DOS manages a tree-structured filing system. We will introduce the concepts of subdirectory and path. We will introduce the five user-level commands related to managing tree-structured systems (CHDIR, MKDIR, PATH, RMDIR, and TREE), show you an example of a set of files that form part of a tree, and, while examining the example, see how these five commands can be used.

A tree-structured filing system is one where directories contain not only entries for files, but also entries for other directories. The first, or root, directory will contain some entries that point to files and some entries that point to other directories. In turn, these other directories can point to a mixture of files and yet other directories. The usefulness of such a structure becomes evident when you have many files to manage. Files that naturally belong together all share (appear in) the same directory. An example showing a typical tree-structured set of files would have to be very large. Here, to demonstrate how to work with trees, we will use as an example only a small portion of a set of tree-structured files.

The Root Directory

The root directory is the primary directory on the disk volume. It is the root of the tree organization.

The root directory is used to create major distinctions between the functions of the groups of files in the tree. It is the only directory that is limited in the number of entries it can hold. (The number of entries is limited to 112 on a 40-track diskette; on a hard disk, the upper limit varies according to the portion of the disk devoted to each MS-DOS disk volume.) It is also the only directory that cannot be deleted.

The entries in the root directory are of three types: a single entry for the volume label, entries for subdirectories, and entries for files. We can ignore the volume label entry (it is not essential to tree structures), and we already are familiar with directory entries for files. Let's look at what a subdirectory entry points to. (By the way, the attribute byte in the directory entry declares the type of the directory entry.)

Subdirectories

A subdirectory is simply a directory that is not the root directory. Frequently the prefix *sub* is omitted. We will often use the word *directory* to refer to both the root directory and subdirectories.

Subdirectories are files that hold a number of directory entries. In practice, there is no limit to the number of entries for files or other subdirectories that a subdirectory can hold.

You can see that when the root directory points to a subdirectory that points to a subdirectory that points to the file of data you happen to be interested in, the operating system and the user will need a path through the forest of directories to find the file.

Paths

A path leads either directly to a file or to an area of the tree that contains files and more directories. The path to a file is the list of the names of the directories, starting with the primary or root directory, continuing through all the intermediate directories that lead to the directory containing the file. The path leads you through the right areas of the tree and ends with the directory containing the entry for the file of interest. The directory names in a path are separated by the backslash character (\).

Now it is time to introduce the five user-level commands for managing tree structures.

Tree-Structure Management Commands

Figure 12.1 shows the five user-level commands that MS-DOS provides for working with a tree-structured filing system.

Let's look at an example of the tree structure in use, to see some examples of paths and to see how these commands are used.

A Sample Tree-Structured Set of Files

We will assume that you have a filing system on drive C that contains several tree sections, each of which serves a different purpose. If you have organized

Command	Function
CHDIR	Change directory
MKDIR	Make a directory
PATH	Define the default path(s)
RMDIR	Remove a directory
TREE	Display the tree structure

Figure 12.1: *Tree-Structured File Management Commands*

your files well, the data that is associated with one activity is not spread out all over the tree, but is isolated in one directory and its subdirectories.

We will examine two areas of the tree: one area devoted to personal budget files, the other devoted to the source-language files of a new game project. Those two groups of files will never be used at the same time, so they belong in different parts of the tree.

There are several files associated with each of the two activities. The budget files include the data files for each category within the budget. The game project files include the source language and compiled programs, the compiler, and some test results. Figure 12.2 shows the tree.

The CHDIR (or CD) Command

When you are working on budgets, your programs will be using the budget directory (and its subdirectories, if any) almost all the time. It would be efficient for the operating system to know that you are mostly going to be operating in one area of the tree, and start its search for the relevant files at the prime directory for that area of the tree.

This establishes the purpose of CHDIR—the change directory command. You use this command to set as your base (current) directory the part of your tree structure where you and the operating system will be most efficient. This directory becomes the current directory. In this part of the tree, the operating system will have to make the fewest disk accesses to get to the files you request, and you will have to type the fewest characters to identify the files you want to work with.

To set the current directory, the CHDIR command (which can be abbreviated to CD) takes a path as an argument. Remember that there is a current directory for each disk volume.

The path from the root directory of drive C to your budget files may be

```
C:\MARY\PERSONAL\BUDGETS
```

The path to your game project may be

```
C:\MARY\BUSINESS\DEVELOP\GAMES
```

The last example demonstrates four levels of directories. To change your directory to the budgets part of the tree, you would type

```
CHDIR C:\MARY\PERSONAL\BUDGETS
```

If you do not remember where in the tree you are operating, the CD command can be used without any arguments to show you the current directory. For example, if you had just started to work on the monthly budget (which you had been putting off all day), and your desperately hungry husband suddenly opened the door and said "Hey, you have to look

after the kid while I eat," and you left the computer to play with your son, then answered a few phone calls, then saw that it was a nice day and mowed the front lawn, by the time you got back to the computer you might not remember the current directory. That is the time to type

```
CD C:
```

You would see

```
\MARY\PERSONAL\BUDGETS
```

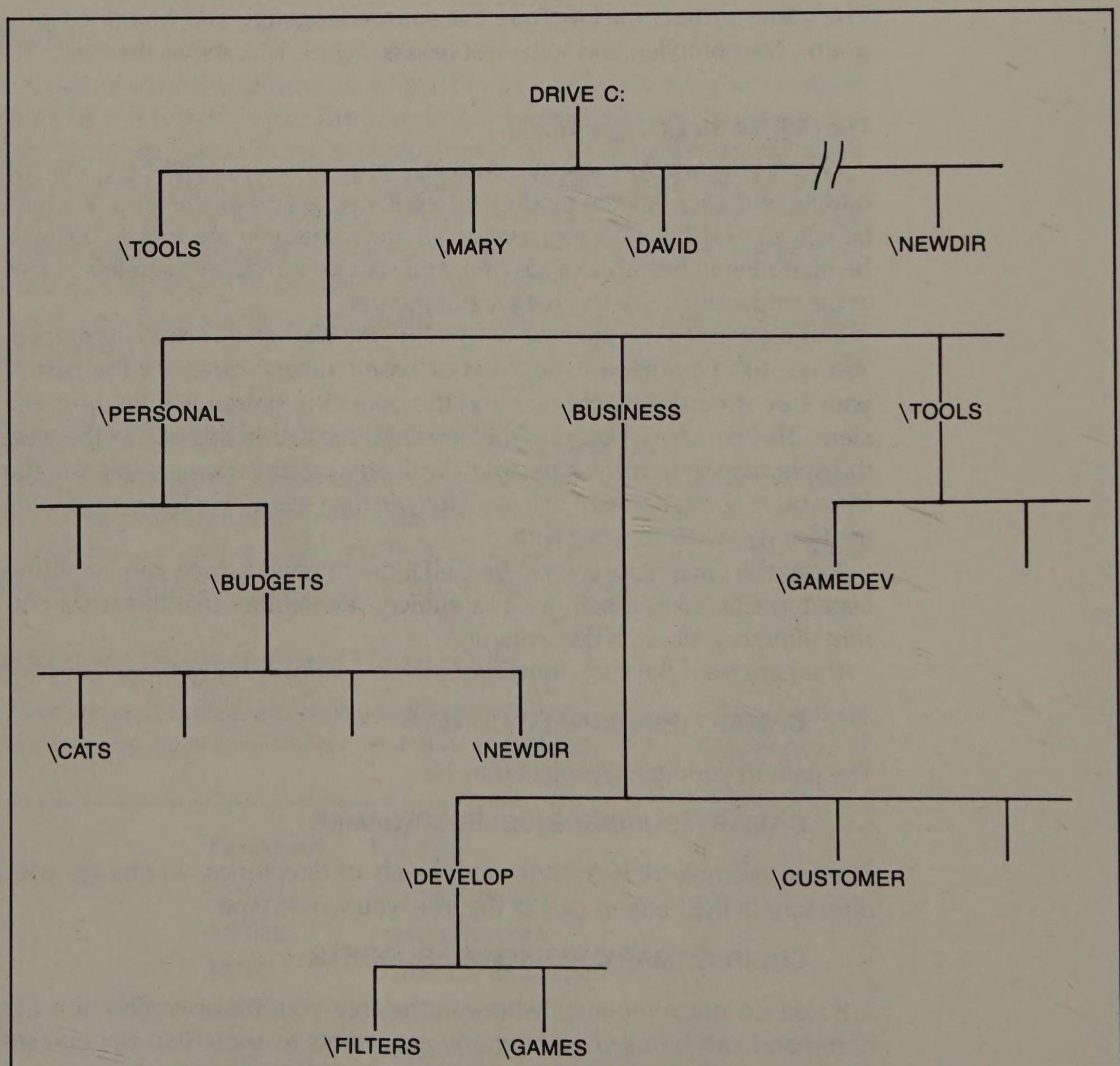


Figure 12.2: *Pieces of a Directory Tree*

At this point you would remember that you had just begun work on the monthly budget.

There is another way to avoid losing your place in a tree-structured directory, and that is to change the MS-DOS prompt so that it always shows the current directory. The command

```
PROMPT $p $g
```

will achieve this.

The MKDIR (or MD) Command

While you are working on the monthly budget, you may decide to create a separate category for all the expenses incurred in getting the local wild cat population fixed so their progeny would not overrun the neighborhood. For this purpose you want a separate group of files, one file per cat. To make the directory, you would type

```
MKDIR C:CATS
```

or

```
MKDIR C:\MARY\PERSONAL\BUDGETS\CATS
```

In the first example, the absence of the backslash following the drive designator tells MS-DOS to make an entry for the subdirectory in the current directory, rather than the root directory.

Removing a Directory—The RMDIR (or RM) Command

Some time later, you may decide that having a separate filing system for the cats is not a good idea. You would incorporate the data from the cat files into existing budget files, and delete the "cats" directory by typing

```
RMDIR C:CATS
```

or

```
RMDIR C:\MARY\PERSONAL\BUDGETS\CATS
```

The Current Directory and the DIR Command

When you are working within a tree-structured filing system for the first time, it is possible to become confused about which directory you see when you use the DIR command. Is it the current directory or the root directory? We will attempt to straighten out that confusion here. The information presented here applies to all the directory-related commands. Each

drive has a root directory and a current directory. (Note: If you do not specify a drive, the default drive—the one you see in the prompt—is used. Also note: The current directory may be the root directory, but since the object of this discussion is to reduce the confusion that results when the current and root directories are not the same, we will ignore the case in which they are the same.)

If you wish to see the current directory, type

```
DIR d:
```

where *d:* is the drive designator. You may omit the *d:* if you want a directory from the default drive.

If you wish to see the root directory, type

```
DIR d:\
```

When the drive designator is immediately followed by a backslash, the command is instructed to ignore the current directory and revert to the root directory. If you type more characters following the backslash, then you are specifying a path to somewhere else in the tree.

Now let's apply this to the MKDIR command. If you wish to extend the directory path beyond the current directory, type

```
MKDIR C:NEWDIR
```

If you wish to add a new subdirectory at the root level, type

```
MKDIR C:\NEWDIR
```

The TREE Command

Here we return from our digression into the difference between a drive's current and root directories, and consider an example where you might need the TREE command.

Some years from now, you may come across a disk with the label "DOS V3—Special Files," and have no memory of the contents. To find out what is on the disk, you would boot MS-DOS version 3, put the mystery disk in the drive, and examine its directory. The fact that some of the directory entries contain the characters <DIR> would prompt you to type

```
TREE A:
```

The tree of directories on the mystery disk would show on the screen. You would perhaps see that there were only a couple of subdirectories, and so use the TREE command again, this time with the /F option, to display the files in the directories as well as the directories themselves.

The PATH Command

The purpose of the PATH command is to instruct MS-DOS where to find the programs and batch files it is asked to execute. If the PATH command has not been executed, MS-DOS expects to find the programs or batch files in the current directory of the default drive.

If you have programs elsewhere in the tree, to execute them you would have to precede the program name with the appropriate path string. The PATH command exists to save you from repeatedly typing path strings, and to save you from having to remember where in the tree each program or batch file exists.

The PATH command establishes a set of directories where MS-DOS might find programs it is asked to execute. Several paths, separated by semicolons, may be specified in the PATH command. The directory at the end of each path is examined in turn when the program is not found in the current directory.

Let's look at an example of how the PATH command can be used. In our previous example, we had two important parts of the tree: the budget section and the game-project section. The path to the game-project area of the tree is

```
C:\MARY\BUSINESS\DEVELOP\GAMES
```

Mary has developed a special set of tools to make game software development more productive. These programs are at

```
C:\MARY\TOOLS\GAMEDEV
```

Before beginning work on game development, Mary would type

```
PATH C:\MARY\TOOLS\GAMEDEV
```

If older versions of the game development tools, which were occasionally needed, were on drive A at

```
A:\TOOLS
```

Mary would type instead

```
PATH C:\MARY\TOOLS\GAMEDEV;A:\TOOLS
```

MS-DOS will look first in the current directory of the drive specified (or of the default drive if none was specified), then it will look in the GAMEDEV subdirectory on drive C, and finally in the TOOLS subdirectory on drive A.

When Mary begins an intense session of work on the game project, and knows that during checkout the system will often crash, she adds the PATH command to the CONFIG.SYS file, so it will be executed every time the

system is rebooted. (Note: after using these commands once or twice, Mary would make her life simpler by moving the commands into batch files.)

Network Paths

In filing systems in a network, the disk drive designator is not necessarily an adequate indicator of where to find a file or set of files. In a network, other computers on the network may be the source of files that you wish to access. A network path could specify the file server computer on which the files exist. MS-DOS uses two backslash characters to indicate the beginning of a local network path. For example, the path

```
\\server2\mary\tools
```

indicates that the tools that Mary will use are to be found on the file server called server2. This convention is expanded to accommodate wide area networks. Assume Mary is part of a large company that uses a wide area network to link the networks in each regional office. Also assume that Mary has just transferred from Miami to Los Angeles. When she first works at the Los Angeles office, she may use the path

```
\\miami\server2\mary\tools
```

to access her software tools.

Let's now look at what you have to do to estimate the maximum capacity requirements of your particular filing system.

File Capacities

Each file can contain a very large number of bytes (for the moment we will use the terms *byte* and *character* interchangeably); in theory, one file can hold more than four billion bytes. A full file containing text can hold about a million and a quarter average-sized pages. In practice, the maximum number of characters is limited by the capacity of the disk. A hard disk volume can contain files of up to 32 megabytes, whereas a file on a floppy disk is limited to about 360K characters.

Planning Your Own Filing System

To create a filing system that won't come crashing about your ears (fingers?) when more is asked of it than you first expected, a little forethought is necessary. It isn't difficult, just a bit of work, and the reward is longer life for your filing system(s) and less repetitive work for you.

In planning for your own filing system, you might consider the following questions:

1. How much data is there going to be in each file?
2. How many files are there going to be?
3. Over how many disk drives would it be convenient for the files to be distributed?

The issues relevant to these three basic questions can be arranged as yet more questions: How much data are you going to accumulate over the life of the filing system? How much data do you need available at any one time? Can you organize the data so that what you need at any one time is compacted into one place? What if the data you need comes in small quantities? Will you have to group it into files, and then have a way to extract one set of data from many in a single file? Can you afford to allocate a single file to each small quantity of data? Finally, how often will files be used, and can they be deleted by any automatic method, such as date triggering, so that your overall capacity is kept down?

You can see from these questions that you have quite a number of ways to go about organizing your files. Let's start by answering the first question.

Estimating capacities can be done by counting characters in an average entry, multiplying that number by the number of entries, and multiplying that by the lifespan of the entries (that is, the number of time periods for which records will be kept). It is vital that you try to think of all the possible items of data you might wish to include. Once you have done this, you are not quite finished. You must double your final result and use that as your guide to total requirements. Once you look at this (probably) large figure, you may be stimulated into putting forth inventive efforts toward making such a quantity of data manageable.

When you estimate expected capacities, think of how long the filing system may be expected to last. It could be that it will grow hugely over time, perhaps to such an extent that you should consider another format for storing the data.

Don't forget to consider the extremes, and realize that your filing system will not necessarily be able to handle every extreme that may occur. Ensure that the response of your system when it comes across one of these extreme situations is reasonable and predictable.

You may want to think of dividing your data among several files. Having additional files is relatively cheap and entails little in the way of overhead, as long as you don't want to switch frequently between the files.

Once you have that doubled capacity estimate, stick with it. Don't be tempted to disregard it when it is not convenient to deal with the difficulties

it creates. There is a tendency not to believe in the need for it when the going gets awkward because of it.

Curtail your urge to be clever and fit everything into a small space. If you give in to that urge, you will be stepping back a generation in computers—back to the 6502 computer, where programmers are for the most part constrained to work within a 64K address space. The 8086/8088 offers one whole megabyte to work with (even though its addressing scheme is burdened with 64K segment limits). You will spend much less of your time trying to cram too much into a small space if you don't succumb to the temptation to think small in the first place. Rewrites caused by lack of space are wasted life.

The big message of this section is that you need to estimate file capacity requirements with as much clarity and honesty as you can muster, then double the estimate and stick with the result. A little pain now can save a lot later; in fact, it may save total failure when you discover how large a rewrite you have to do to meet your "unexpected" capacity needs. By the time you have done the rewrite, the whole project may have missed its deadline and been superseded.

Here are two final notes before we go on to choosing file names. The first recommends frequent use of the CHKDSK command; the second is on minimum file sizes and the risk of wasting space.

Use CHKDSK Often

There is a useful MS-DOS command, CHKDSK, that tells you the space available on a disk. Use it to keep an eye on disks that you use at all frequently. Too many applications programs don't handle a lack of disk space with any aplomb at all. Some are downright brutal—they don't even give you the chance to change disks. They may waste hours of your work when they can't immediately do what you want by way of storing data on a disk. (Note: Do *not* use version 1 CHKDSK on version 2 or 3 disks.)

A Note on Minimum File Sizes

Each time a file expands, more space is allocated to it from the disk (assuming any is available). This space is allocated in fixed amounts, not just the quantity currently needed. The amount of space allocated depends on the type of the disk. On single-sided 40-track disks, the space is allocated in units of 512 bytes, or one sector. On double-sided disks, the space is allocated in units of 1024 bytes, or two sectors. (Both of these units are called clusters by the file allocation routines.) You need to keep these quantities in mind when planning your filing system. It may be that each of your files will need 1100 bytes to store the required data. That means that each file will be about twice as large as it needs to be, and that you will run out of disk space

sooner than expected. The situation can be worse on a large-capacity fixed disk where space may be allocated in chunks of 2048 or even 8192 bytes.

Choosing File Names

One tool you have to work with in organizing files is the file name. MS-DOS uses an eight-character filename with a three-character filename extension. Judicious assignment of meaning within those two fields can lead to names of files that are easy to remember and use, and that describe, with very little need for interpretation, the contents of the file.

Let's look at file names in a little more detail before we go on to talk about choosing them carefully. We'll first explain the difference between file name and filename, then see how long the file names are in a directory entry, then show the character set from which file names can be made. The last part of this diversion into detail is about MS-DOS's reserved file names.

Distinction between File Name and Filename

The name of a file in MS-DOS terminology consists of a "filename" of one to eight characters, followed immediately by a period and a "filename extension" of one to three characters. The period and filename extension may be omitted.

In this book, *file name* means the name of a file, *filename* refers to the characters before the period, and *filename extension* refers to the period and the characters following it. The distinction may seem pedantic, but it is useful. Also, you often will see "extension" used instead of "filename extension."

Number of Characters in a File Name

We've said that a filename can be made up of one to eight characters. That may be true when you type it, but a filename in a directory entry will always contain eight characters; a filename extension will contain three. If you don't supply all eight (or three) characters, the field will be padded out to the right with spaces to make up the correct number of characters.

Permissible Characters in MS-DOS File Names

MS-DOS provides a broad set of characters that can be used in file names. They are as follows:

- The letters A–Z (if you type lowercase characters, they will be converted to uppercase)
- The digits 0–9
- The special characters ! @ # $ % ^ & ( ) _ - ~ { } ' '

You can see that some characters cannot be used. Following are the characters, mostly mathematical and common punctuation symbols, that cannot be used in file names:

* + = [ ] : ; " , . ? / ¦ < > \ space, tab, and any control characters

Now that we have seen the fiddly details of what goes into an MS-DOS file name, we can look at how MS-DOS reserves some file names for special purposes.

Spaces in Filenames

MS-DOS won't let you imbed a space in a file name, but you may see such a name in a directory. The chances are that BASIC created the file. If you want to delete or rename a filename with an imbedded space, use BASIC.

Reserved File Names in MS-DOS

MS-DOS reserves for its own use some specific file names. It is not compulsory that you avoid the use of these names at all times, but using them without understanding their purpose is risky. Files could disappear without trace, replaced by something quite unexpected. Once you understand what a reserved name is about, however, you can sometimes use it to advantage.

Some commercial applications software adopts certain file names or extensions for its own purposes. For instance, if you were editing a large file with the WordStar program and your work was terminated by a power failure, you would later see a file with an extension of .$$$. Like EDLIN, WordStar uses *XXX*.$$$ for the name of a temporary file, where *XXX* is your filename for the file. Independent software writers usually stick with MS-DOS conventions, though they sometimes branch out into their own inventions. If you're lucky, that information is available in their documentation.

Some manufacturers' MS-DOS manuals emphasize the use of trailing colons to distinguish between device names and file names. They say CON: is the console device, and allow it to appear that CON (without the colon) might be an acceptable filename. With or without the colon, however, CON means the console device, and may not be used as a filename. A table of the MS-DOS reserved file names is provided in Figure 12.3.

Now let's finally get to some suggestions for choosing file names.

Choosing Names with Care

There aren't too many brilliant ideas here, just a few common-sense approaches to selecting file names that are clear and unambiguous, and notes on some temptations to avoid.

Reserved Filenames

Filename	*Meaning*
%PIPE	Temporary files for data in a pipe
@ . . .	Any filename beginning with the character @; used by the auto response file in the linker
BADTRACK	Created by version 1.0 to hold addresses of unreadable clusters
FILE*nnnn*.CHK	Lost clusters recovered by CHKDSK
FILE*nnnn*.REC	File names resulting from the whole disk RECOVER process
NUL	Directs the file to the bit bucket, i.e., to nowhere retrievable
VM.TMP	Created by the linker when there is not enough room in memory for the load module
device name	Device names cannot be used as file names

Reserved Filename Extensions

Extension	*Meaning*
.$$$	EDLIN, or other, temporary file
.ASM	Assembler source file
.BAK	Backup file, deleted and created by EDLIN
.BAS	BASIC program source file
.BAT	Batch file
.BIN	Binary image file
.COM	Command file
.CRF	Cross reference source file
.EXE	Executable file, produced by the linker
.HEX	DEBUG assumes this is a hex ASCII file and converts it to binary on loading
.LIB	Source code library files
.LST	Listing file from the assembler
.MAP	Linker's map
.OBJ	Assembled object file
.REF	Cross reference listing file
.REL	Relocatable file from the assembler
.TMP	Temporary file

Figure 12.3: *MS-DOS Reserved File Names*

Assigning names to files becomes simpler when you choose to limit your options by adopting file-naming conventions. The conventions must not be unacceptably restrictive; they can be chosen so that the gains they offer far outweigh the disadvantages. For example, people have adopted the convention of driving on the right-hand side of the road (in this country). It is a restrictive convention, but driving is a considerably less messy experience because of it. So it is with file names. And here you can make up your own set of rules, not have others' imposed on you.

When you choose file names for a set of files with a common purpose, varying the extension systematically can be enough to distinguish the members of the set. For instance, you can use the Julian date for the extension. That fits into three digits, varying from 1 to 365 (or 366 on leap years). The Julian date can be a little hard to decipher, which makes it not the best choice for file names that will be interpreted by people; a program, however, would make sense of it with no trouble.

If you need a file that is specific to a single date, using that date for the filename is an option. It is hard to mistake the meaning of the filename 9-15-86. You need to be consistent in your inclusion of leading zeros, and you could omit the separator characters, thus leaving the other two characters of the filename available for indicating the type of file of that date.

For groups of files with a common purpose, placing them in a single directory is a good start. To differentiate between them, in addition to varying the extension, sometimes you may also need to vary one or two characters of the eight-character filename to get enough variety. It is better to vary the "low-order" characters than the leading characters. The wild-card (* and ?) options that MS-DOS makes available for selecting file names are a little easier to apply that way.

Ambiguity creeps into file names because at the time the name (or set of names) is chosen, whoever is making the choices is deeply involved in his or her project. It is easy to forget to step outside for a fresh look at what you are doing, and so miss the ambiguity of a name like *vest*. In an inventory context, *vest* means an article of clothing; in an employee-pension context, it may refer to the amount vested. (You may be justly unimpressed by the aptness of this example; in your experience ambiguous names will occur with more facility.)

Choose meaningful names—not just names that are meaningful to you, but to your grandmother as well if she wanted to run your filing system. Don't abbreviate file names, except for those of a few, very frequently used files. Abbreviated names seem to be clarity itself when they are invented, but they often turn out to be thoroughly confusing when viewed in another context. The choice of meaningful names applies to more than just short

names. Even long names can be seen by another to mean something quite different.

Mnemonic names can be useful, but not a unique and inconsistent extraction of certain "unnecessary" characters. A phonetic approach is usually doomed—the variety of pronunciation in this country is too great.

Simple truncation is often adequate. Exercise your powers of language, even use a dictionary of synonyms to come up with alternatives to ambiguous names. This can also be entertaining for whoever works with your filing system.

Then there is the secret approach. This involves making all names totally indecipherable. This is not infrequently achieved by accident, occasionally by design. You need enthusiasm and a good memory for this approach to work. Don't forget that a year from now you may need the data in your files, and that you may have become absorbed in an entirely unrelated field.

Optimum Placement of Files on the Disk

You might need the quickest possible execution time in some filing system applications. If you don't have enough memory left for a *ram disk* (a section of memory that is made by software to look like another disk drive), small improvements in performance can be gained by optimum placement of the files on the disk.

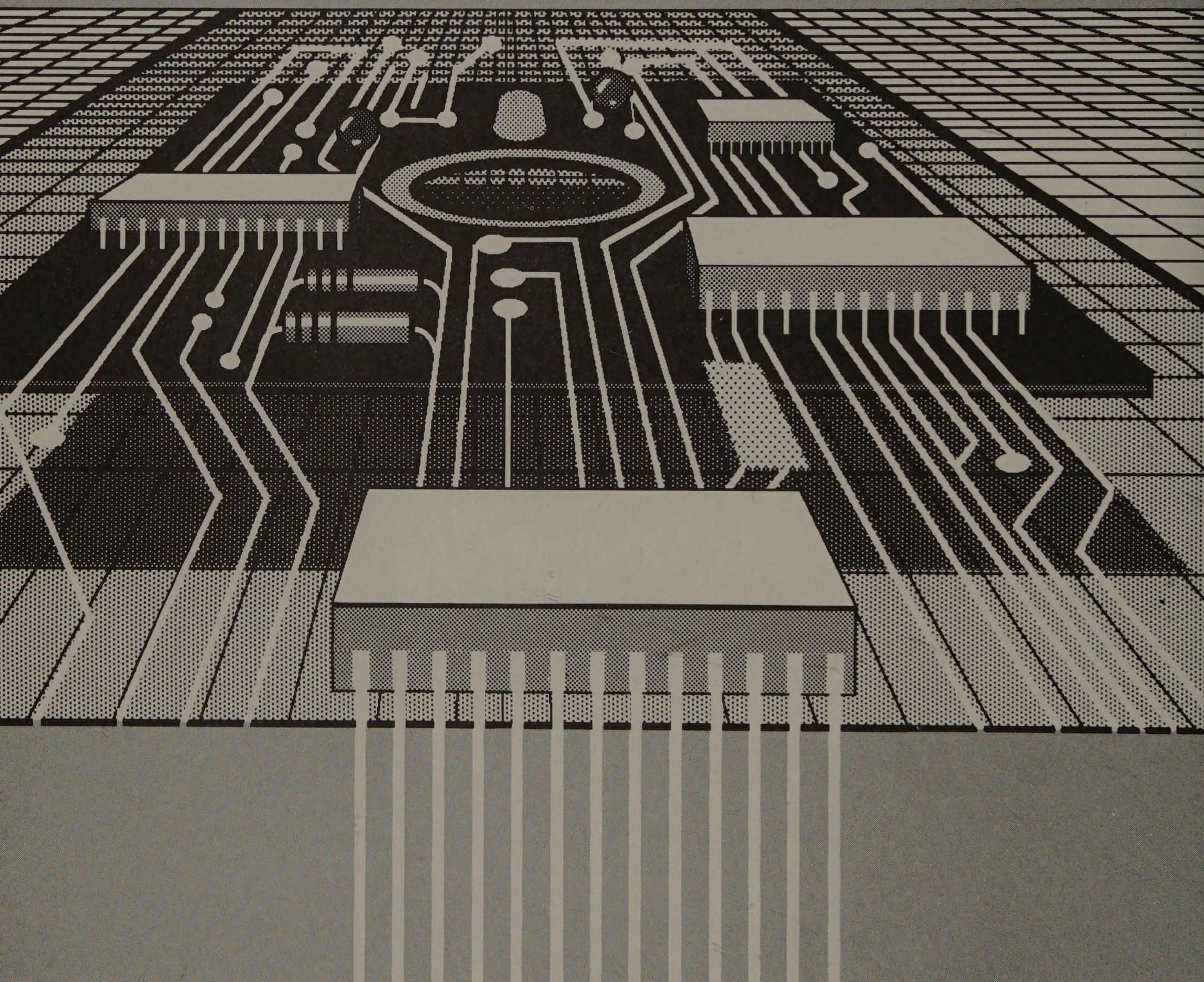
Small files that are to be accessed alternately should be kept next to one another, and close to their directory. Use the COPY command (not DISKCOPY), to shuffle files to the right places on the disk. On systems without a hard disk, consider keeping data that belongs in two large files on two separate disks, with one in each drive (if you have a two-drive system) for quicker alternate access.

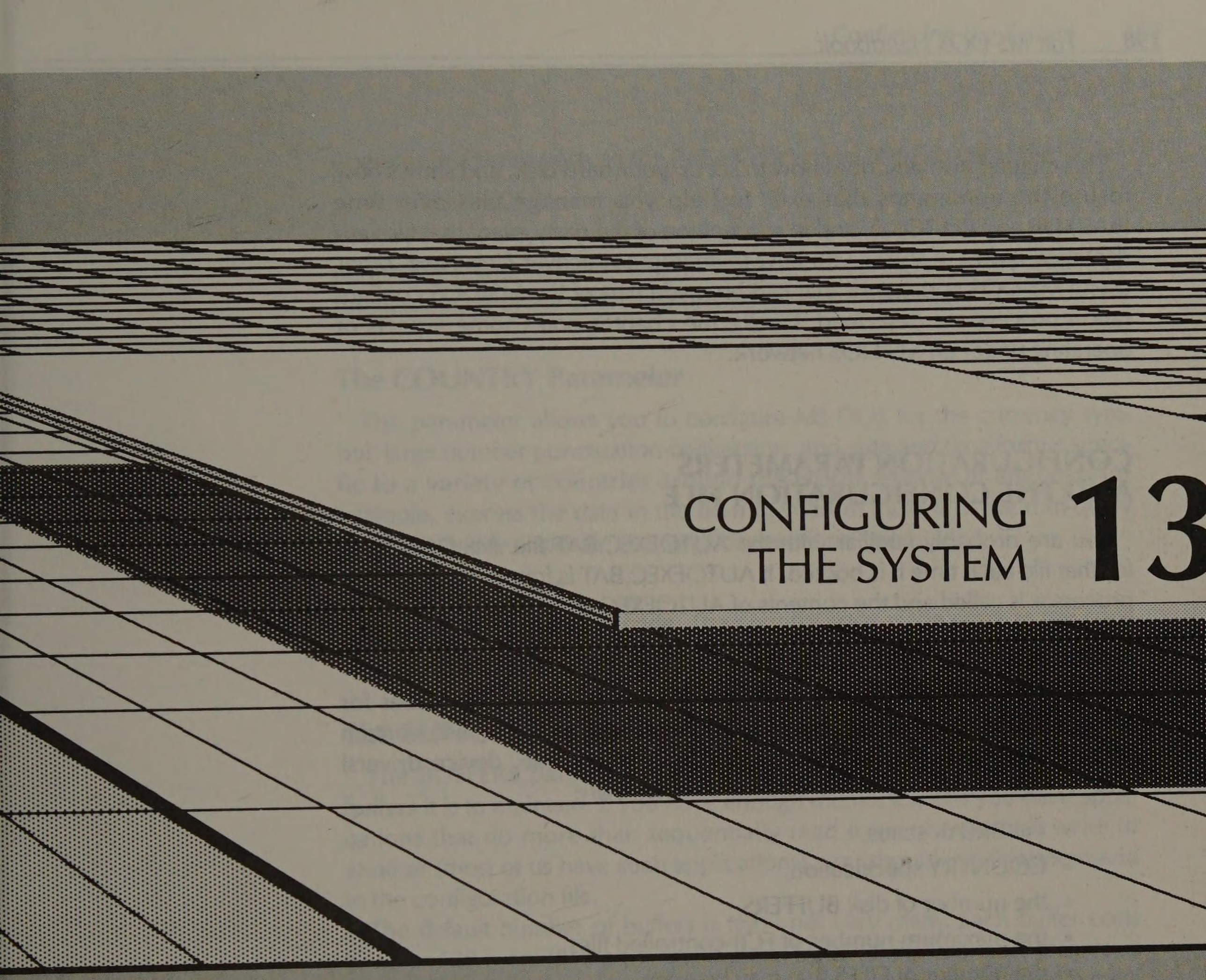
Placement of clusters within files on the disk is controlled by a disk-space allocation mechanism. When an MS-DOS file expands, the file allocation routines step through the file allocation table looking for the first free block (or cluster) to allocate to the file.

When a file shrinks, any extra clusters are returned to the pool of free blocks on the disk. Over time, as files on a disk expand and contract, the clusters within a file will occupy indeterminate positions on the disk. To clean up these files and put them in order, so that each file occupies consecutive blocks on the disk (and thus is open for the best rate of access), use the COPY command. Make up fresh disks by using FORMAT (without the /S option), then use COPY *.* B: (where the newly formatted disk is in drive B:). This will reorder the clusters within your files. The COPY command can be used with files one at a time to relocate those files that need to be kept together.

SUMMARY

We have seen how to make best use of the file-management system MS-DOS supplies. We have seen that your choice of file names can make the difference between confusion and clarity, and that, on occasion, the physical location of files on a disk may affect performance. MS-DOS uses a tree-structured filing system that is easy to manage and is limited only by the capacity of the disks containing it.





CONFIGURING THE SYSTEM 13

INTRODUCTION

MS-DOS allows you to specify how the operating system is to be configured (set up). This chapter shows you how to set up and maintain your system, and how to make use of these configuring tools so that you can tailor your system to perform well in its task of meeting your specific needs. The configuring tools consist of parameters that are set at the time the system is booted, and commands that can be executed also at boot time, or later. The configuration parameters are described first, then the configuration commands.

This chapter also describes how to set up your hard disk, and shows how to use the commands that exist to help you manage files over time (BACKUP and RESTORE), and in recognition of the nasty event that files are damaged or lost, it shows you how to attempt to recover files. Next, other useful MS-DOS commands that can be made part of the MS-DOS configuration process are described. Finally, the chapter notes some aspects of operating under an MS-DOS network.

CONFIGURATION PARAMETERS AND THE CONFIGURATION FILE

You are probably familiar with the AUTOEXEC.BAT file. MS-DOS looks for that file each time it is booted. If AUTOEXEC.BAT is found, the batch file processor is called and the contents of AUTOEXEC are executed. However, well before MS-DOS looks for the AUTOEXEC.BAT file, it looks for another file—CONFIG.SYS.

CONFIG.SYS is the configuration file. If it exists, it contains values for some or all of the configurable MS-DOS parameters. It can also contain instructions for building additional features (for example, device drivers) into the operating system. The configurable items are

- the BREAK status,
- COUNTRY specification,
- the number of disk BUFFERS,
- the maximum number of FCB-controlled files,
- the number of FILES that may be open,
- the maximum number of disk drives,
- the SHELL,
- environment variables, and
- additional DEVICE drivers.

The parameters are specified by adding to the CONFIG.SYS file strings of the form

```
BUFFERS = 10
```

We'll examine these parameters in turn.

The BREAK Parameter

The BREAK parameter can be used to change the frequency with which MS-DOS checks for the Break character (Ctrl-C, or Ctrl-Break [PCCs]). It

operates in a similar way to the BREAK command. Adding the string

```
BREAK=OFF
```

to the CONFIG.SYS file turns extensive break checking off. If no BREAK parameter is specified, the system comes up in the BREAK=OFF state. See the description of the BREAK command for more information.

The COUNTRY Parameter

This parameter allows you to configure MS-DOS for the currency symbol, large number punctuation convention, and date and time format specific to a variety of countries around the world. People in England, for example, express the date in the dd-mm-yy form instead of the mm-dd-yy form. To specify a specific country's conventions, use

```
COUNTRY nnn
```

where *nnn* is the country code. The country code is the same as the code used in international telephone calls. Not all country codes are supported.

The BUFFERS Parameter

The BUFFERS parameter allows you to tell MS-DOS how many disk buffers it is to maintain. If you have enough memory, and if you have applications that do more than sequentially read from one file and write to another (most of us have such applications), you should use this command in the configuration file.

The default number of buffers is two—not very many. Each buffer costs another 528 bytes (512 plus 16 overhead bytes). Performance increases as you add more buffers, but at somewhere between ten and twenty buffers performance begins to decrease again because it takes the operating system as long to look through all the buffers as it does to get something from the disk. The difference between having two and five buffers is very significant. The difference between having five and ten buffers is less significant, but still a noticeable improvement. Assuming your system is not short of memory, reserve space for ten buffers by adding the string

```
BUFFERS=10
```

to the CONFIG.SYS file.

The FCBS Parameter

When file sharing is in operation in current versions of MS-DOS, space is reserved for only four concurrently open FCB-controlled files. The string

```
FCBS=10,5
```

tells MS-DOS to allow for up to 10 concurrently opened FCB-controlled files, and protects 5 of them from automatic closure by MS-DOS. MS-DOS will close the least-recently-used FCB file to make way for the opening of another FCB file. This parameter can be useful in network operations when you are running older applications software that still uses file control blocks (FCBs) instead of the newer file handles. If you see error messages that mention FCB, add the FCBS command to your CONFIG.SYS file.

The FILES Parameter

MS-DOS can handle the details of eight different files being open at the same time (these are file-handle files, not FCB files, and include the three always-opened standard input and standard output devices). The number provided in the FILES parameter is a system-wide number. The per-process maximum of 20 files simultaneously open cannot be changed.

An application that uses several files, running in conjunction with the background PRINT function, and running on a network can strain the default capacity of MS-DOS to handle simultaneously open files. When you need more, the FILES command comes to the rescue. With it, you can tell MS-DOS to annex the resident space required to keep track of up to 99 files. To tell MS-DOS to accommodate 25 different simultaneously open files, to the CONFIG.SYS file add

```
FILES = 25
```

The LASTDRIVE Parameter

The highest drive letter that MS-DOS will recognize is E, unless you set the value of this parameter higher. On networks, where a variety of disk volumes are available to you, using

```
LASTDRIVE = J
```

allows five more disk volumes to be addressed.

The SHELL Parameter

If you have written your own command processor, called COMPROC-.COM, inserting the line

```
SHELL = COMPROC.COM
```

in the configuration file will cause MS-DOS when it is booted to call COMPROC.COM instead of COMMAND.COM.

The DEVICE Parameter

This parameter is used to tell MS-DOS to include device driver software for devices that are not normally supported by MS-DOS. You may have a ramdisk, or a tape unit, or even 8-inch floppy disks attached to your PC. The chances are that the device will need special software to control it, and that that software may be supplied by the device manufacturer. All you have to do is add the line

```
DEVICE = TAPEDVR
```

to the configuration file, where the file TAPEDVR has been supplied by the tape unit manufacturer, and your tape accessing programs that run under MS-DOS will be able to operate the tape unit.

The extended cursor control and keyboard key reassignment features are included in one such device driver, ANSI.SYS. The ramdisk feature is included in VDISK.SYS. If you want these extended features, the lines

```
DEVICE = ANSI.SYS
DEVICE = VDISK.SYS
```

must be added to CONFIG.SYS.

ANSI stands for American National Standards Institute. Amongst other standards, the institute provides perhaps the most versatile standard for cursor positioning, and Microsoft has adopted it for MS-DOS. If you include the ANSI.SYS file as a device driver, you have control of cursor positioning and keyboard key reassignment at a far simpler level than whatever could be kludged together in MS-DOS version 1. See Chapter 4, "MS-DOS and the Keyboard," for keyboard key reassignment, and see Chapter 5, "Output to the Display Monitors," for the form of cursor control enabled by the presence of the ANSI.SYS device driver.

CONFIGURATION COMMANDS

The configuration commands, like the ability to change configuration parameters, allow you to change both hardware and software features of MS-DOS without having to dig around in the hardware or turn into a programmer. Configuration commands are typically used in the AUTOEXEC.BAT file, but some of them can be used at any time to allow you to adapt to new situations. The commands we describe here are

- MODE or CONFIG,
- BREAK,
- PROMPT, and
- SET.

The original configuration command is MODE [PCCs] or CONFIG.

The MODE (or CONFIG) Command

The first incarnation of the MODE command allowed the user to configure the serial port's parameters and to redirect the parallel printer output to a serial port. (See Chapter 16, "Using the Serial Port," for more information on setting the baud rate, parity, number of data bits, and number of stop bits.) Different manufacturers add features to the MODE command. A common option exists for switching between the different display adaptors.

Display Adaptor Selection

In the IBM PC and its compatibles two display adaptors are offered, the monochrome and the color/graphics (see Chapter 15 for more information). The display width is 80 characters on the monochrome adaptor, but may be either 40 or 80 on the color/graphics adaptor. The default on powering up the system is 80-character lines on a black-and-white display. The command to set a display mode is

MODE *dmode*

where *dmode* is chosen from Figure 13.1.

The first six options are effective only if you have a color/graphics adaptor. If you use only a monochrome adaptor, the single option avaiable is the last, monochrome.

Image Alignment on the Display

On some displays the image will not be centered on the screen. The MODE command has the ability to horizontally shift the image on a display

dmode	Display Mode
40	40 characters per line
80	80 characters per line
bw40	40 cpl, black and white
bw80	80 cpl, black and white
co40	40 cpl, color
co80	80 cpl, color
mono	80 cpl, monochrome

Figure 13.1: *Display Mode Options*

screen so that it fits better. The format for this is

```
MODE dmode,direction,T
```

where *dmode* is optional and has the meaning shown in Figure 13.1; *direction* is either R or L, depending on whether the image needs moving to the right or left. The T parameter, if specified, produces a test pattern that makes alignment selection easier.

Printer Mode Selection

Another common feature of the MODE command is to specify the format for the printer. The choices are 80 or 128 characters per line, and 6 or 8 lines per inch vertical spacing. The format of this variant of the MODE command is

```
MODE LPTn:cpl,lpi,P
```

where *n* specifies printer 1, 2, or 3; *cpl* is either 80 or 128; and *lpi* is either 6 or 8. The default values are 80 characters per line, 6 lines per inch. The P is optional and tells MS-DOS to continuously retry if the printer gives time-out errors.

Key Press Feedback Control

On some MS-DOS computers, including the COMPAQ, audio feedback to a key press is via the computer's speaker (on the IBM PC, there is tactile feedback and mechanically produced audio feedback from the key itself). The COMPAQ's MODE command includes an option to adjust the volume of the audio response (a click) to a key press. The format is

```
MODE cli,n
```

where *n* is value from 10 to 127. On some COMPAQ models the feedback volume may be increased using the Ctrl-Alt+ combination and decreased using Ctrl-Alt-.

Another configuring command is the BREAK command.

The BREAK Command

The BREAK command is almost identical to the Break parameter. It can be used to change the frequency with which MS-DOS checks for the Break character (Ctrl-C, or Ctrl-Break [PCCs]). Normally only a few function calls recognize the Break character. However, when BREAK=ON is in effect, MS-DOS checks for Break when *any* function call is executed. Typing

```
BREAK
```

shows the current break status. Adding the ON or OFF arguments, as in

```
BREAK ON
```

or

```
BREAK OFF
```

turns extensive break checking on or off.

A program operating under MS-DOS makes function calls. "Making a function call" means the program makes use of the MS-DOS built-in tools that get characters from the keyboard, access the disk, and perform other commonly needed operations. When BREAK is ON, *any* function call to MS-DOS by the program will cause MS-DOS to check for, and so respond to the presence of, the Break character. When BREAK is OFF, only a few MS-DOS functions (screen, keyboard, serial port, and printer I/O) respond to the presence of the Break character.

BREAK OFF is the normal mode of operation. Using BREAK ON can help you explore programs that are new to you. One of the last items of information provided in most program documentation and in many help files is how to quit from the program. It's rather shortsighted of the program author/documentor not to provide this information sooner. It seems they assume that a user has much time and an unquenchable desire to learn every last detail about a program before discovering the exit button. Using BREAK ON before exploring a new program may help.

The response of some programs to the Break character is unpredictable (as far as the user is concerned), and some programs override the Break mechanism. You learn by experience whether you can benefit from using BREAK ON with a particular program.

The PROMPT Command

The appearance of MS-DOS to the user may be changed by using PROMPT to alter the MS-DOS prompt message. This message is what you see when MS-DOS is ready for a command. Instead of the usual C> or A>, you can have appear the time and date; an anthropomorphic, chatty greeting (such as Hi Honey, What'll it be?); the version number; the current directory (path); or all of these in combination with a number of other possibilities.

The PROMPT command is of the form

```
prompt strings
```

where *strings* is a sequence of regular text and what IBM calls meta-strings, a code that selects an item to be included in the prompt. A meta-string is a

single-character code that follows the $ sign (to add the $ sign to the prompt text, use the string $$). For example

```
PROMPT Current directory is $p on $d$g
```

provides the prompt

```
Current directory is C:\ on Tue 9-16-86>
```

The meanings of the meta-strings are shown in Figure 13.2.

Prompt has a second use because of its ability to deal directly with the Escape character. The ANSI control sequences for keyboard key reassignment and cursor control all begin with the Escape character. The rest of MS-DOS assumes an Escape character means "cancel the current command line and start a new one," which has the effect that strings containing the Escape character cannot be entered from the keyboard. However, PROMPT can be used to send the ANSI control character strings. See the sections on keyboard key reassignment and cursor control in Chapters 4 and 5 for more information.

Using PROMPT to alter the prompt string is fun to play with, but it's hard to come up with a prompt that is more than merely informative yet retains its charm after several hundred repetitions.

Meta-Code	Item Added to Prompt
t	Current time
d	Current date
n	Default drive letter
p	Current directory (of default drive)
v	Version number
g	>
l	<
b	|
q	=
h	Backspace
e	Escape
_	New line (carriage return and linefeed)
$	$

Figure 13.2: *Meta-String Codes in the PROMPT Command*

The SET Command

Use the SET command to pass arguments to applications programs that look into their MS-DOS environment.

The SET command accepts and displays the strings found in the current MS-DOS environment. You can enter whatever strings you choose; they don't have to be strings that are understood by MS-DOS. Programs can look into that environment and take action that depends on what they find there. For example, the environment contains the current PATH through the tree structure (MS-DOS puts it there whenever it changes). This PATH is used by the command processor to guide its search for programs to execute.

The environment is implemented as text strings in a buffer within the command processor. The strings are entered and deleted using the SET command. To see the strings in the environment, type SET without any arguments. To enter a string (that, say, describes a selected accounting method) into the environment, type

```
SET A/CMETHOD=CASH
```

To remove that same string from the environment, type

```
SET A/CMETHOD=
```

Chapter 14, "Batch Operations," gives an example of using the SET command to put a string into a batch file's environment, where it can be used to replace a dummy argument.

HARD DISKS AND FILE MANAGEMENT

The hard disk preparation and management commands are described in this section, as well as the command that helps recover files when disaster strikes. The preparation for loading MS-DOS onto a hard disk is handled by the FDISK and FORMAT commands. The commands that back up and restore are, surprisingly, called BACKUP and RESTORE. The file recovery command is RECOVER. Each of these commands is examined.

The FDISK Command

FDISK is one of the few MS-DOS commands that offers a menu. Four or five options are available for working with disk partitions. For each disk drive partitions may be created, changed, deleted, or reviewed. In IBM terms those options are install, modify, remove, and list. A partition is a portion of the disk that is devoted to a particular operating system. Most MS-DOS users use the whole disk for the MS-DOS partition, but you may

want to divide the disk space into, for example, one area (partition) for UCSD Pascal and the rest for MS-DOS.

Follow the menu instructions offered by the FDISK command as necessary. On the assumption that you are primarily an MS-DOS user, allocate as much disk space to the MS-DOS partition as you can afford. Disk space gets used quickly. When making partition changes, remember to save any useful information in a partition before making changes that will destroy it!

The FORMAT Command

Beware of the FORMAT command. Any disk, floppy or hard, that you format will lose all data that used to exist on it before the format operation. Unlike file deletion, there are no commercially available programs that will allow you to recover old data from a freshly-formatted disk.

The command to format a newly-partitioned hard disk is

```
FORMAT C:/S
```

or

```
FORMAT C:/S/V
```

if you wish to add a volume name to the formatted disk.

The BACKUP Command

The BACKUP command must be used on a regular basis to ensure that in the event of a catastrophic hard disk failure, for example, a head crash, you will be able to recover quickly. The schedule of backup is up to you, but don't wait to learn that backup is necessary until after the first time you discover how useful the backup feature could have been.

BACKUP takes four different options. You must specify the disk (floppy or hard) to which the backup copies will be written. In addition you can choose to back up all files that have been changed since a certain date. The command

```
BACKUP C: A:/D:9-14-86
```

backs up from drive C onto drive A all files that have been modified since 14th September, 1986. All previous files on drive A will be lost. If you wish to add to files on drive A instead of replacing files, use

```
BACKUP C: A:/D:9-14-86/A
```

The command

```
BACKUP C:*.* A:/S
```

will back up all files, including those in subdirectories (the /S option), onto the disk in drive A. If additional disks are required, BACKUP will ask for them. If you run BACKUP on a regular basis, the command you will most likely use is

```
BACKUP C:*.* A:/S/M/A
```

which adds to an existing backup disk only those files anywhere in drive C that have changed since the last BACKUP was run.

The RESTORE Command

This command is used when you wish to retrieve one or more files from a backup disk. For example

```
RESTORE A: C:TEST2.*
```

will restore from the disk in drive A all files that have a filename of TEST2. If you want to be selective about which files are restored, use the /P option, as in

```
RESTORE A: C:TEST2.*/P
```

RESTORE has the complement operation to a full disk backup. The command

```
RESTORE A: C:/S
```

will restore all files from the disk(s) in drive A to drive C, including all subdirectories and their files.

The RECOVER Command

The RECOVER command recovers data, as best it can, from damaged files. It can be directed to work on either a single file or a whole disk. RECOVER can retrieve undamaged data as long as one file allocation table is intact.

Recovery of Single Files

RECOVER will recover the remaining data from a file that has one or more unreadable sectors. The format for this operation is

```
RECOVER A:MYFILE
```

where MYFILE is the name of the file you want recovered.

When the RECOVER command has finished, you will probably need to clean up what's left with an editor, for two reasons: (1) the data in the unreadable sector(s) will not be in the recovered file; and (2) RECOVER sets

the file length in the directory to the number of clusters (alias allocation units) in the recovered file. This will be a multiple of 512, and it is unlikely that, say, a text file would contain an exact multiple of 512 characters.

You have to call the RECOVER command for each file you want recovered. Although the command will accept wild-card characters (* and ?) in the file name, it will only recover one file before it quits.

Recovery of Whole Disks

This option is used when the directory is damaged. The format of the call is

```
RECOVER A:
```

This wholesale recovery operation is quite drastic. In the whole-disk-recovery mode, RECOVER steps through the file allocation table, looking for chains of clusters. When it finds a likely looking chain, it makes an entry in the directory for a file by the name FILE*nnnn*.REC, where *nnnn* is 0001 for the first file found, 0002 for the second, and so on.

RECOVER on a whole disk should not be lightly invoked. It is for desperate people. If the disk has tree-structured files on it, the tree will not exist after the recovery operation. All the files will be in a single directory.

ASSORTED USEFUL COMMANDS

A few other MS-DOS commands may have been briefly mentioned elsewhere in this book but deserve a longer description here. Some of them can be made automatic and so become part of the MS-DOS configuration process. These are the commands:

- PRINT
- GRAPHICS
- JOIN
- SUBST
- ASSIGN
- CTTY
- COMMAND

Background Printing with the PRINT Command

The PRINT command will accept a list of files to be printed, put them in a queue, and print them as a background task while other programs are

being executed. Options within the PRINT command allow you to

- specify the printer to be used (PRINT /D:device),
- display the queue (PRINT without any arguments),
- add more files to the queue (PRINT file name(s)),
- delete files from the queue (PRINT file name /C),
- terminate the whole printing-from-queue operation (PRINT /T),
- specify the number of files the print queue can hold (PRINT /Q:n), and
- alter the performance of the background queue printer.

If you are already familiar with the PRINT command in MS-DOS version 2, you will notice welcome additional "tuning" features in version 3.

The queue by default holds a maximum of ten files at one time. The command to increase the queue size to say, 20 files, is

```
PRINT/Q:20
```

The maximum number that can be specified is 32 files in a queue.

Background PRINT Performance

Version 3 adds the ability to tune the performance of the background PRINT command. The available parameters are buffer size, timeslice size, maximum per-character "tick" allocation, and maximum wait for printer-ready condition.

Buffer Size The default internal print buffer size is 512 bytes. Increasing that size makes the PRINT operation more efficient. For example

```
PRINT/B:1024
```

increases the buffer size to 1K.

Timeslice Size The default timeslice allocated to background printing is eight clock ticks. The command

```
PRINT/S:16
```

doubles the time allocated to background printing. The range of possible values is 1 to 255.

Maximum Ticks Per Character The command

```
PRINT/M:4
```

allows the PRINT function four clock ticks to print a character—twice the normal value. The range of possible values is 1 to 255.

Maximum Wait If your printer is intermittently slow to respond to a character, but usually available to print a character, and you don't mind a slight slowdown of the foreground program, you might want to use the command

```
PRINT/U:3
```

How you adjust these parameters depends on the nature of the foreground task that runs in conjunction with the background PRINT, and on the performance characteristics of the printer to which the files are directed. You may find it necessary to have batch files established that specify the PRINT parameters that, by experimentation, you have found best for each of your common PRINT uses.

When you add files to the queue in version 3 they don't have to come from the current directory; you can specify a path to the file. In earlier versions if you wanted to print files from different directories, you had to change to the directory containing the file before adding it to the queue.

Of course, you may not use the printer for other purposes while the PRINT function controls the printer. Also, the first time the PRINT command is called, about 3600 bytes are added to resident MS-DOS.

The PRINT command can be called from the AUTOEXEC.BAT file without user intervention by using, for example,

```
TYPE CRLF ¦ PRINT/Q:15/B:1024
```

where CRLF is a file containing nothing but a single carriage-return linefeed.

GRAPHICS to the Printer

This is a nice feature to have. It prints on a dot-matrix printer an image of the screen, sideways if necessary. (In the IBM PC and its compatibles, it is useful when PC-DOS is operating with a color/graphics adaptor. It has no relevance when PC-DOS is operating with the IBM monochrome adaptor.)

Invoking the GRAPHICS command has two effects: (1) resident MS-DOS is increased by about 700 bytes, and (2) when operating in any graphics mode, the print-screen-image keystroke prints the image on the screen driven by the graphics adaptor.

Where the quantity of pixels on the screen is greater than can be printed across one printer line (for example, where the IBM color/graphics adaptor is in 640 × 200 mode) the image is printed sideways, with the image rotated 90 degrees counter-clockwise. The bottom of the screen appears on the right-hand side of the paper.

The JOIN Command

JOIN is used to join the files contained in a disk drive volume with an empty directory on another volume. After the JOIN, all the files in the

volume are available in the directory specified. Giving the command

```
JOIN E: C:\fredsvol
```

means that all the files in E can be referenced via the directory C:\fredsvol. To undo this join, use

```
JOIN E:/D
```

The SUBST Command

The SUBST command allows you to fool a program that does not recognize path names into using them indirectly. SUBST substitutes drive letters for other drive letters or for paths. The command

```
SUBST D: E:\working\tools
```

will allow a program that can address drive D but cannot use paths to get to the files in \working\tools. To discontinue the substitution established in this example, use

```
SUBST D:/d
```

A variety of MS-DOS commands won't work correctly while SUBST is in effect. This is another not fully implemented command that will probably be improved upon in forthcoming releases. In the meantime, perhaps ASSIGN can help.

The ASSIGN Command

The ASSIGN command allows you to fool some drive-specific software into operating on another disk drive. It redirects disk I/O requests from one disk drive to another. The command

```
ASSIGN B=C
```

will allow programs that want to use drive B to use drive C instead. To disable the effect of the ASSIGN command, use ASSIGN without any arguments.

The CTTY Command

CTTY is useful for directing MS-DOS to accept its commands from another device, such as a terminal linked to your computer via a serial port. To do that, type

```
CTTY COM1
```

Once the CTTY command has been executed, it is the new device that must

give the command to return to normal operation. The original keyboard and screen are disabled. The command to return to the regular console is

```
CTTY CON
```

Secondary Command Processors

MS-DOS has the ability to invoke a secondary command processor. (It also has the ability to operate with a command processor of your own choosing. See the description of the SHELL parameter in the section on configuration parameters.) A secondary command processor can be called upon to execute MS-DOS commands from within a program or a batch file. Invoking the secondary command processor takes up about an additional 4K or 5K of memory, but that is not expensive compared with the gains to be had.

The COMMAND Command

COMMAND allows you to invoke a secondary command processor. The secondary command processor will see and interpret what is entered at the keyboard (or the current standard input device) instead of the regular command processor. To leave that secondary command processor and return to the original (or previous) command processor, use the EXIT command.

Chapter 14, "Batch Operations," shows how a second copy of the command processor may be invoked in the middle of a batch file; this can be used to produce nested batch files.

USING MS-DOS IN A NETWORK

If your PC can become part of a network, you need to know how to link your PC into the network, and to think about which of your resources you are willing to share with the other users on the network. A complete description of networks is outside the scope of this book, but the essentials are presented here. For an excellent and complete description of the IBM PC Network, see *Operating the IBM PC Networks* by Paul Berry (SYBEX, 1986).

The hardware required for a PC to join a network is a network adaptor card. This adaptor is installed in your PC and connects to the cable, coaxial or other type, that connects all computers on the net.

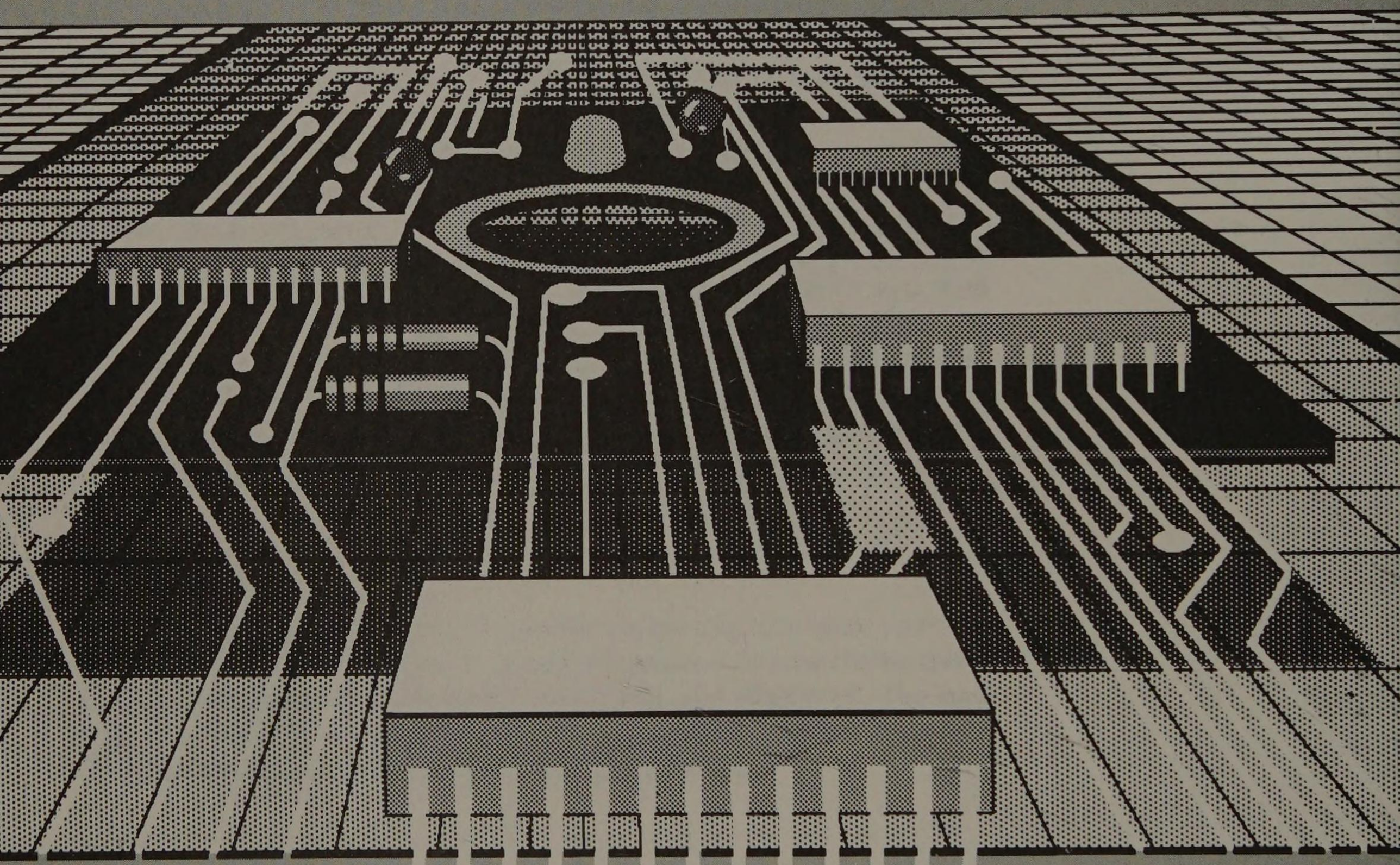
MS-DOS doesn't have all the facilities required to manage a networked PC, so additional software, for example, the IBM PC Network, is required. This software is loaded after MS-DOS is up and running. It directs the adaptor in its task of putting data into the network's data stream at the right times, and manages the interaction between MS-DOS and the network.

MS-DOS does have some network-related features. File sharing is one of them. Executing the SHARE command lets MS-DOS know that it must deal with situations where more than one user may be accessing a single file. Like the PRINT command, SHARE puts additional code in memory. This SHARE code intercepts all file accesses and ensures that conflicting requests are handled correctly. Sharing must be installed (simply by entering the SHARE command) before network operations can begin.

The IBM PC Network program offers menus of commands that let you choose how to participate in the net: what to offer for sharing, how to link to what's available elsewhere. Once you have declared that a device of yours, say, a printer, may be shared by others on the net, then you must access it via the net. After you have offered to share your printer, you too must send requests for output to that printer through the regular network channels and not attempt to use the printer directly. For more information on the concepts involved when MS-DOS is part of a network, see Chapter 9, "Networking with MS-DOS."

SUMMARY

We have reviewed the MS-DOS commands that help configure your MS-DOS system to your specific needs, and have seen how to use the hard-disk management tools FDISK, BACKUP, RESTORE, and RECOVER. The new features of the PRINT command have been described, and we have seen the basics of network operations and network concepts.



BATCH OPERATIONS 14

INTRODUCTION

This chapter will help you become familiar with using batch files. It will show you what they are, what they can do for you, and how they work in MS-DOS. In addition, it presents examples showing how to create batch files.

A batch file contains the commands for a sequence of operations. Once you have created such a file, MS-DOS will work away at those commands while you get coffee, call your broker, or get ready for bed at 2 a.m. (when you want to back up your night's work so you can sleep in peace).

MS-DOS will execute a sequence of commands in a batch file, instead of waiting for you to type them in one by one. If you don't like repetition, the batch file facility is just what you need.

Batch files, by the way, are *not* the path to background processing or multitasking in MS-DOS—that capability is not currently available. Multitasking allows you to have more than one task running at the same time. Attractive though that is, MS-DOS does not yet offer it.

APPLICATIONS FOR BATCH FILES

Batch files have two major areas of application: (1) they give you freedom to walk away from the computer while it performs a sequence of operations unattended, and (2) they serve as a place for storing a (perhaps) complex sequence of commands that you want performed without any being missed.

Batch files can be used for establishing consistent startup procedures and for defining sequences of operations for backup at the end of a day. They are also useful for issuing the sequence of commands that directs output to a printer with a serial interface, or establishes a connection via modem with Dow Jones News/Retrieval Service, Delphi, or some other on-line information service. In general, it is worthwhile to enter any sequence of oft-repeated commands into a batch file.

In addition, if your typing skills are not the best and/or you are repulsed by the effort involved in typing the names of certain commands (e.g., CHKDSK A:), you can use batch files to alleviate your pain by abbreviating such commands (e.g., CHKDSK A: to CKA).

Batch files can contain messages and prompts, to guide you through non-trivial operations and thereby ensure that your performance is flawless.

The most often used batch file is AUTOEXEC.BAT. It supplies a set of commands executed any time the system is booted. By the way, if you create your own AUTOEXEC.BAT, do remember to include the DATE and TIME commands, or the command that reads your computer's clock.

HOW BATCH FILES WORK

In brief, the command processor calls the batch-file processor whenever it sees a command which is a file name that ends with the letters .BAT. Let's expand on that a little.

MS-DOS contains a set of routines for recognizing commands. This set is called the command processor. It also contains a set of routines for processing batch files. This group of routines is called the batch-file processor.

The command processor looks at the characters that are typed in response to the system prompt (usually C> or C:). It sees these characters as soon as you press the Return/Enter key. The command processor first tries to match the characters with the resident MS-DOS commands. If there is no luck there, it then looks for a file on disk whose name matches what has been typed.

If you do not include a filename extension (a period followed by one to three characters), the command processor makes a first guess that the extension is .COM and looks for that. Its second guess is .EXE, and its third and last guess is .BAT, which is where we come in. If the command processor finds the .BAT file, it passes control to the batch-file processor.

The batch file processor has enough logic in it to read the first line (a command) from a batch file, invoke the command, and, when that terminates, read the next command from the batch file. This process continues until the batch file is exhausted or the user aborts the sequence.

Note: If the command you type in response to the system prompt contains a filename extension, the command processor will look only for files with that specific extension; the batch file processor won't be invoked unless that extension is .BAT.

As far as the user is concerned, the command processor gets a string of characters just as easily from a .BAT file as it does from the keyboard.

Aborting Batch File Operations

You can abort either the current command that has been called by the batch processor or the whole set of commands waiting to be executed.

To abort batch file operations, type the Break character. If the program being executed responds to that character, you will see the following message:

```
Terminate batch job (Y/N)?
```

This is your chance to either terminate only the current command or quit the whole batch sequence. If you want to forget the current command and move on to the next in the batch file, type N, (meaning no, you don't want to quit the whole sequence). Type Y to terminate the whole sequence.

There is a fly in the ointment when terminating batch jobs. The current command (program) being executed may not recognize the Break character. You will discover this after waiting patiently for a few seconds after typing the character and seeing no 'Terminate batch job (Y/N)?' message. Here you have a chance to exercise your knowledge of the program being run; give it the command that normally makes it quit. After you have done that, you should get the terminate message. Then you can proceed as usual, giving a Y or N answer.

Because termination of batch job sequences depends somewhat on the good behavior of the program being executed, the results of aborting batch job sequences are not always predictable.

Dummy Arguments

There is an important additional feature within the batch processor—dummy arguments. To make a batch file's range of applications more flexible, dummy arguments can be indicated using the percent sign (%). The dummy arguments are filled in at the time the command they are associated with is executed.

Nine unrestricted dummy arguments are available. They are %1 through %9. The examples of how to create a batch file expand upon the use of dummy arguments, show you the purpose of the %0 dummy argument, and more. See the section "More on Dummy Arguments" later in this chapter.

CREATING A BATCH FILE

A batch file is a file containing one or more lines of text. That's all there is to it. You can create a batch file using EDLIN or another editor, as you would create any file of text. You can also use the COPY command.

You might use the COPY command to create a simple batch file that you are sure does not already exist, or that you don't mind overwriting. Complex batch files, however, will usually have to be edited, so you might as well be in the editor initially for these files.

For demonstration purposes, we will use the COPY command method here; its format is

```
COPY CON XXX.BAT
```

where CON is the device name for the console, and XXX is a filename of your choice, up to eight characters. Here are several examples of batch-file creation using the COPY command.

Example 1—Automatic Program Execution

On the IBM PC*jr's* version of MS-DOS (PC-DOS version 2.1), a head settle time of 15 milliseconds has been allowed for in all disk transfers. On the IBM PC the read heads stay in contact with the floppy disk; on the PC*jr* they do not, hence the need for the head-settle time. A program could be written to change the head-settle time to zero (there is a parameter table in the disk routines) if the operating system (that is, version 2.1) were running on the IBM PC instead of the PC*jr*. We could call that program SPEEDUP. Because the modification to speed up the disk accessing has to be installed

every time the version 2.1 PC-DOS is booted on a non-*jr* PC, the popular solution is to include a call to SPEEDUP within AUTOEXEC.BAT, the batch file that the system executes (if it can find it) on startup.

Assuming that no AUTOEXEC.BAT file already exists (if it does, use an editor to add lines 2, and perhaps lines 3 and 4, of the following sequence), this sequence will cause SPEEDUP to be called on system startup:

```
COPY CON AUTOEXEC.BAT
SPEEDUP
DATE
TIME
<Ctrl-Z>
```

On the IBM PC compatibles, the F6 key may be used instead of Ctrl-Z, unless the meaning of the F6 key has been reassigned. Also, CON: may be used instead of CON.

Note: *Beware of using the COPY command to create an AUTOEXEC.BAT file.* Any such file that already exists will be overwritten—a lot of good work may be lost. Once a complex or pleasing AUTOEXEC.BAT file has been created, you might keep a copy of it in AUTOEXEC.SPR (not .BAK).

Example 2—Abbreviation of a Command

To abbreviate the CHKDSK command to CK, enter this sequence:

```
COPY CON CK.BAT
CHKDSK
<Ctrl-Z>
```

Now, each time you type CK, the CHKDSK program will be invoked. To check a disk, assuming it is in drive A, use

```
COPY CON CK.BAT
CHKDSK A:
<Ctrl-Z>
```

Example 3—Abbreviation with Dummy Arguments

The dummy arguments %1 and %2 are used here. To abbreviate the RENAME command to the letter R, use

```
COPY CON R.BAT
RENAME %1 %2
<Ctrl-Z>
```

If you then type

```
R BILL WILLIAM
```

the batch-file processor will replace %1 with the first argument, BILL, and %2 with the second argument, WILLIAM. The command processor will thus see

```
RENAME BILL WILLIAM
```

and the file called BILL will be renamed WILLIAM.

Let's look at dummy arguments in a little more depth before moving on to the next example.

More on Dummy Arguments

Recall that nine dummy arguments are available, %1 through %9. In addition, %0 will always be replaced with the filename of the batch file; this is useful for repeating an operation indefinitely, as shown in the next example. The SHIFT command allows you to use more than nine dummy arguments, and thanks to the SET command, you can create batch files that take their dummy arguments from the environment. See "Dummy Arguments from the Environment" later in this chapter.

Example 4—Repetitive Action with Prompt

The REM and PAUSE commands can be included in a batch file to describe the action of the file to the user and to serve as a prompt when one is required. To see directories of several disks without having to call for the directory each time, create a batch file by typing

```
COPY CON DIRS.BAT
PAUSE Load next disk into drive a:
DIR A:/P
%0
<Ctrl-Z>
```

This batch file displays the directories of a series of disks placed in drive A. Note that the batch file is passing control, in its last statement, to itself. It is "daisy chaining," not calling, so stack overflow is not a concern. The /P option tells the DIR command to pause at the end of each page of a directory listing.

When the batch-file processor sees a REM statement in a batch file, it displays the text of the REM statement and goes on to process the next line of the file. If the batch-file processor encounters a PAUSE statement, the text of the PAUSE statement is displayed, followed by the message

```
Strike a key when ready
```

The batch-file processor waits until a key is struck before processing the next command in the file. In the example above, the user would strike a key

after inserting another disk in drive A. Now that we have seen a few simple examples of batch facilities, let's look at the commands that allow you to write more complex sequences of operations for MS-DOS.

FOR, IF, GOTO, SHIFT, AND ECHO

We will look at these commands one by one, then go through a sample batch file to see how these commands can be used. We will then see how we can make one batch file call another as a subroutine.

FOR Loops in Batch Files

In BASIC and other languages, FOR loops typically operate by incrementing a variable at the end of each iteration; they terminate when the variable reaches a specified value.

The batch FOR command takes its variable from a list included in the FOR statement and terminates when the list is exhausted. The result is the same in both cases—repeated execution of the same command. A sample FOR command is

```
FOR %%y IN (1983 1984 1985 1986) DO Summary %%y
```

where Summary is a program that takes the year (%%y) as an argument.

In the batch version of the FOR loop, the instruction that follows the word DO can contain a single command. This one command can be another batch command (except FOR), a program to be executed, a request to execute, another batch file as a subroutine, or a DOS command. The FOR loop cannot contain another FOR command.

Another sample FOR command is

```
FOR %%e IN (2 3 4 5) DO IF ERRORLEVEL %%e GOTO Exit
```

The IF Command

With the IF command, you can test for three different types of conditions: (1) that two strings are the same (IF *string 1* == *string 2*), (2) that a command or applications program returned an error condition equal to or greater than a given number (IF ERRORLEVEL 5), or (3) that a file exists (IF EXIST *filename.ext*). The example following the ECHO command description (listed in Figure 14.1) shows several uses of the IF command.

The GOTO Command

The GOTO and IF commands, together with the facility for returning numbered error indicators from applications and some systems programs,

turn the batch-command processor into an interpreter of a minor programming language.

GOTO goes to the line following the label. (A *label* is a line beginning with a colon.) The example that follows later shows the use of a GOTO command, and how labels can be used to document the batch file. The text of the label is the characters (no more than eight) that follow the colon, up to the next space or the end of the line.

The SHIFT Command

If you need more than nine dummy arguments (also called dummy parameters), the SHIFT command lets you get at those beyond nine. It can also serve as a way to process an indefinite number of arguments. The SHIFT command moves the arguments down each time it is used; for example, the fifth argument becomes the fourth argument after the shift command is executed. The %*n* dummy argument becomes the %(*n* – 1) argument, and the %0 argument is lost.

The ECHO Command

The batch-file processor displays the commands in the batch file as it executes them. The ECHO command, which takes the arguments ON and OFF (and message), can be used to suppress that display. Batch files that contain the ECHO OFF command appear cleaner in operation; they don't muddle the screen with extra chat.

If ECHO is OFF, the messages associated with the REM and PAUSE commands are also suppressed (although you do see the "Strike a key . . . " part of the PAUSE command). You may still want some messages to appear, which is where the ECHO message command comes in. Whatever the ECHO status, the message part of an ECHO message command will appear. For example, where the batch file SHOWECHO.BAT contains:

```
REM  This batch file demonstrates the ECHO command
ECHO  OFF
REM  This message will not appear
ECHO  Echo is off
ECHO  ON
REM  Echo is now on
```

typing SHOWECHO at the system prompt (say, A>) will produce:

```
A>REM  This batch file demonstrates the ECHO command
A>ECHO OFF
A>Echo is off
A>REM  Echo is now on
```

Thanks to I/O redirection, the ECHO command can be used to write information into a file. For example, the lines

```
ECHO Data of interest > \WORKING\DATAFILE.EXT
ECHO More data >> \WORKING\DATAFILE.EXT
```

will first create the file DATAFILE.EXT with the line

```
Data of interest
```

in it, then add to that file (using >> instead of >) the line

```
More data
```

Let's examine a sample batch file that includes the SHIFT, IF, GOTO, ECHO, and FOR commands.

A Sample Batch File

Here is the example. The name of the file is SHOWDUE.BAT. It is invoked by typing

```
Showdue (Month) Martha Jason
```

where (Month) is one of the months of the year. For example:

```
Showdue June Martha Jason
```

The purpose of the file is to enable someone who runs a day-care center (with the help of two assistants, Martha and Jason) to see if the monthly bills for each child in her care have been paid. The file is run with the feature of MS-DOS in effect that echoes output to the printer line by line (Ctrl-PrtSc in the IBM PC compatibles, Ctrl-P (on) and Ctrl-N (off) in some others).

The file SHOWDUE.BAT is shown in Figure 14.1. A sample run of that file is shown in Figure 14.2; another sample run is shown in Figure 14.3.

This batch file is far short of being an elegant and practical program, but it does serve to illustrate batch file capabilities. Let's examine it.

The owner of the day-care center celebrates Christmas by not charging her clients for the month of December. Line 2 takes care of that; the resulting output is shown in Figure 14.3.

The owner (Angelica Sanchez) is planning to add more assistants, so she has used the SHIFT command in line 6 to step through all the assistant's names entered on the command line. Each SHIFT instruction moves the next assistant's name into the %1 parameter. Line 7 will end the batch file operation if there are no more assistant's names. If your version of MS-DOS does not support this test, a file for each assistant (*xxxx*.ast, where *xxxx* is the assistant's name) can be created so that the line

```
IF NOT EXIST %1.ast GOTO End
```

```
ECHO OFF
IF %1 = = December GOTO FreeMnth
ECHO This is the %0 list for the Month of %1
ECHO The following children's %1 bill(s) have not been paid:
:NextAsst
shift
IF x%1 = = x GOTO End
ECHO %1's Group:
GOTO %1
:Martha 's group
FOR %%%c IN (Jonathan Mary Joy Joey) DO IF NOT EXIST
  %%%c.pd ECHO %%%c
GOTO NextAsst
:Jason 's group
FOR %%%c IN (Ashley Grendl Susan Hud) DO IF NOT EXIST
  %%%c.pd ECHO %%%c
GOTO NextAsst
:FreeMnth
ECHO This is the free month!
:End
```

Figure 14.1: *The SHOWDUE.BAT File*

```
A>Showdue June Martha Jason

A>ECHO OFF
This is the Showdue List for the Month of June
The following children's June bill(s) have not been paid:
Martha's Group:
Jonathan
Mary
Joy
Jason's Group:
Ashley
Grendl
Susan
```

Figure 14.2: *Output of the SHOWDUE.BAT File*

can be used to determine when the list of assistant's names has been exhausted.

Lines 7, 8, and 9 each show a different use of the %1 parameter.

In lines 10 and 13, the rest of the line beyond the label (Martha in line 10, Jason in line 13) has been used to make each of those lines a comment—in the batch file. (The temptation to be slightly clever was irresistible.) Note the space between the end of the label and the "comment." That space terminates the label, and in this example makes the line look a little odd.

The FOR instructions in lines 11 and 14 look for a file with a filename, consisting of a child's name and an extension of .pd (for example, JOEY.PD), for each child in the group. Each month, before running this batch file, Angelica creates the .pd files for each child whose monthly bill has been paid.

Now let's look at Figure 14.2, the output of this batch file. For the month of June, only Joey's and Hud's parents came up with the day-care money on time.

In this example, only a few variations on the use of batch commands have been demonstrated. It can be fun to experiment with others.

Calling One Batch File from Another

Chaining from one batch file to another is no problem. However, returning to the calling batch file from the called file necessitates invoking a second copy of the command processor. Let's look at each of these methods.

Chaining from One Batch File to Another

To terminate execution of one batch file and begin execution of another, simply place the name of the second batch file in the last line of the first batch file. The batch-file processor will obediently call and execute the second batch file. Note that the name of the second batch file is always the last line executed in the first file; any lines following are never processed. For

```
A>Showdue December Jason Martha

A>ECHO OFF
This is the free month!
```

Figure 14.3: *The Christmas Season Output of SHOWDUE.BAT*

example, the batch file EUROPE.BAT contains

```
COPY B:BELGIUM.* A:
COPY B:FRANCE.* A:
COPY B:SWITZ.* A:
MALAYSIA
```

and MALAYSIA.BAT contains

```
COPY B:SINGAPOR.* A:
COPY B:INDONESI.* A:
COPY B:MALAYA.* A:
```

After the batch processor has copied all your European files, it will abandon the EUROPE.BAT file and copy all your Malaysian files. Then it will return control to the command processor, which sends the DOS prompt to the screen and waits for input. Changing EUROPE.BAT to contain

```
COPY B:BELGIUM.* A:
COPY B:FRANCE.* A:
COPY B:SWITZ.* A:
MALAYSIA
COPY B:HOLLAND.* A:
```

would *not* get the Dutch files copied.

Calling One Batch File as a Subroutine from Another

In this example, we make up a batch file FOREIGN.BAT that contains the following:

```
rem This file copies all foreign customers' files
COPY B:UK A:
COPY B:IRISH
COMMAND/C EUROPE
COPY B:TURKEY A:
COMMAND/C MALAYSIA
rem Operation complete
```

We use a version of EUROPE.BAT that looks like this:

```
COPY B:BELGIUM.* A:
COPY B:FRANCE.* A:
COPY B:SWITZ.* A:
```

Typing the filename

```
FOREIGN
```

at the DOS prompt will copy files from a total of nine different countries.

The extra copy of the command processor that is brought into memory (for copying the European and Malaysian files) needs about 4K (or so) bytes. The COMMAND/C operation is not implemented identically in all versions of MS-DOS. If you have trouble using it, try another version of MS-DOS.

DUMMY ARGUMENTS FROM THE ENVIRONMENT

The SET command allows you to specify the content of the DOS environment. The *environment* is a set of strings that applications programs can find and refer to. The strings can therefore control the operation of the program, or even a batch file. The batch file can look into the environment for a dummy parameter. Let's demonstrate this with an example.

Assume that we want a batch file that types a description of the vegetable of the week, where each week presents a different vegetable. We want to leave the batch file unaltered from week to week. Each week we would put the new vegetable into the environment by typing, at the DOS prompt, a command like this:

```
SET Vegetable = Asparagus
```

This enters the string

```
VEGETABLE = Asparagus
```

into the environment. We would have created the batch file VEGOWEEK.BAT as follows

```
COPY CON VEGOWEEK.BAT
rem This file types a description of the vegetable of the week
type %VEGETABLE%.des
Ctrl-Z
```

Description files (whose names are of the form VEGFILE.DES where VEGFILE is the name of the vegetable) have been prepared for a variety of vegetables. Once the vegetable of the week has been entered into the environment, at any time we could type

```
VEGOWEEK
```

to see a description of this week's vegetable. Note the use of bracketing percent signs in the line

```
type %VEGETABLE%.des
```

to tell the batch processor to pull the dummy argument from the environment.

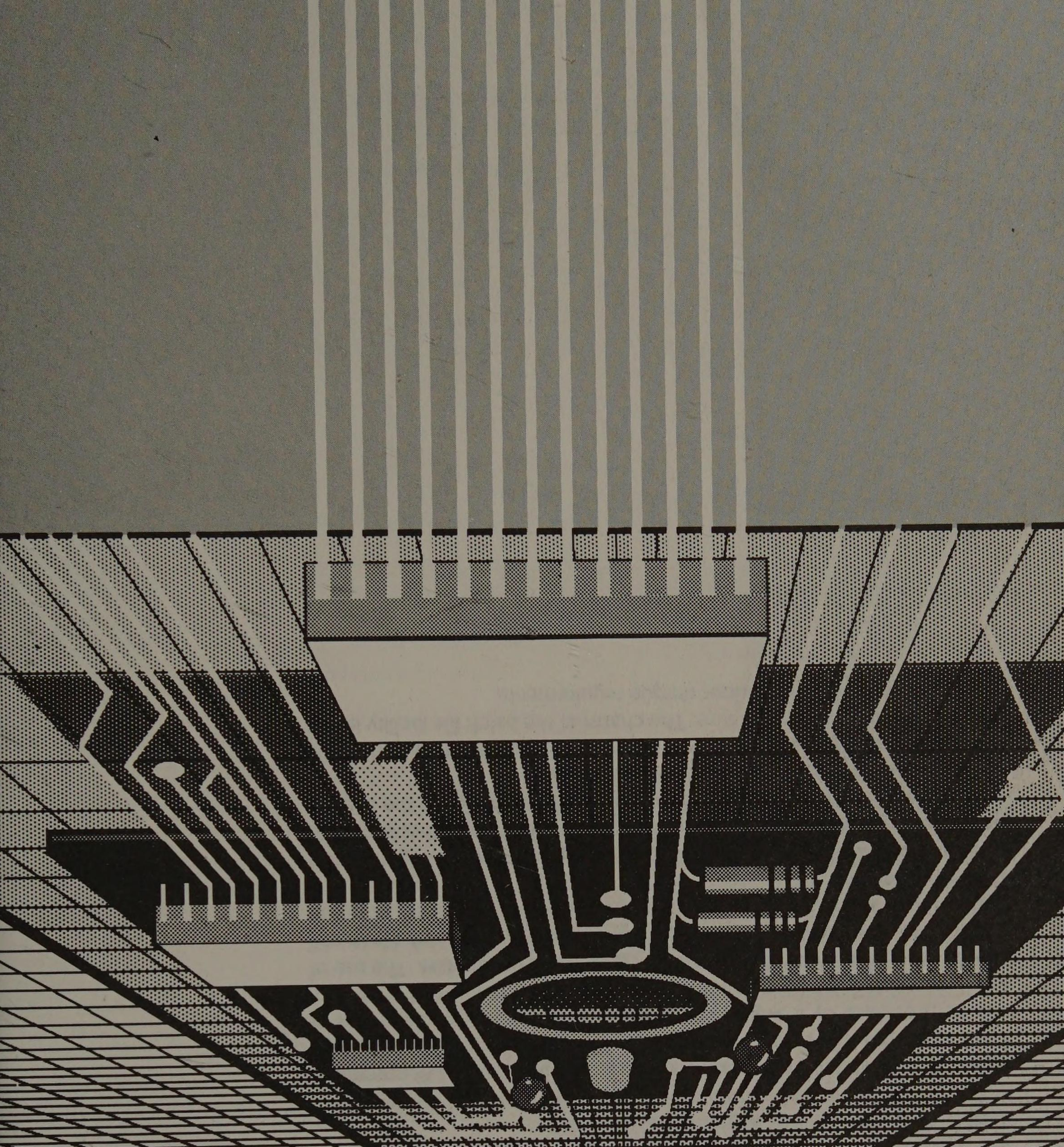
SUMMARY

We have seen how batch files succeed in their purpose of making life easier for you. A single batch file can simply abbreviate a command, or it can control long and complex sequences of operations.

The ECHO command allows us to add remarks without cluttering the output of the batch file operations. The %% variable in the FOR command can include the wild-card characters * and ?, which means that many files can be processed by a single FOR command. The string-matching part of the IF command can be a handy feature. The SHIFT command gives us a way to process an indefinite number of replaceable parameters. The use of a secondary copy of the command processor enables one batch file to call another as a subroutine. Dummy arguments may be obtained from the DOS environment.

Do use this facility for disk backup. Figuring out how to organize the batch file for backup will help you organize your disk files so that backup is relatively simple. This in turn will encourage you to actually *do* the backup operation, which may save you some later agony.

Keep the batch-file facility in mind. It can make your use of your MS-DOS computer simpler and more sure. The charm of the batch file facility is that it can be tailored to your various unique requirements.



THE MONOCHROME AND COLOR/GRAPHICS SCREENS

15

INTRODUCTION

This chapter is about the ways your MS-DOS computer sends characters to the display, and how the appearance of those characters may be controlled. It's one of the areas where the technology is still rapidly advancing, so each manufacturer's selection of displays and video capabilities may not be the same as another's.

Many manufacturers follow the leads established by the IBM Personal Computer, where a high-resolution monochrome display is offered along with displays capable of showing graphics as well as individual characters. A display that can show graphics is one where each dot on the screen may be individually addressed.

This chapter describes the capabilities of computers that match the video options provided by IBM in its Personal Computer. Many MS-DOS computers provide the same options, although your computer may be different. The information provided here will serve as a guide to the relevant factors when you consider video displays.

MS-DOS computers use memory-mapped displays, where characters that appear on the display are found in memory. The display adaptor reads from memory to decide what characters to send to the screen. When a program changes a location in memory that is part of the display buffer, the character on the display immediately changes.

A display requires special circuitry to drive it. Often that circuitry is provided not by the computer's main system board, but by an adaptor that is suitable for driving a particular type of display. It is the adaptor that the program addresses, and that controls the appearance and resolution of the characters on the display.

The video options that we examine are:

- the monochrome display and adaptor
- the color/graphics adaptor (any display) and
- the enhanced graphics adaptor.

The chapter begins with a description of the monochrome display and the adaptor associated with it; we then move on to the broader subject of the color/graphics and enhanced graphics adaptors and the different types of displays they can drive. The two graphics adaptors operate in two modes: alphanumeric (A/N) and graphics (APA—all-points-addressable). We examine both modes. A section on alphanumeric character output in graphics mode is provided. The somewhat odd procedure for color selection is described, and tables listing the colors available in the text and graphics modes are provided. The chapter ends with comments on choosing a monitor and a section on why not to match randomly selected adaptors with randomly selected monitors.

You may want to know how to program the output of characters to the screens, and how to choose the right character attributes to achieve the desired color, reverse video, blinking, or other effects. Read this chapter first for the necessary background, then refer to Chapter 5, "Output to the Display Monitors."

THE MONOCHROME DISPLAY

The display device that offers excellent character resolution is based on the IBM Monochrome Display, a high-resolution, high-persistence green phosphor monitor. This monochrome monitor has a sufficiently high resolution (720 × 350) to support a pleasing and clear character set, displayed in 25 lines of 80 characters each. The characters are made from a 7 × 9 matrix of dots within a 9 × 14 box. The monochrome monitor is addressed via the monochrome display adaptor.

THE COLOR AND GRAPHICS CAPABILITIES

The path to both color and graphics displays is usually provided entirely independently of the monochrome capability. Two adaptors are offered that can be used for both color and graphics. These adaptors can be purchased in addition to, or instead of, the monochrome display adaptor. Many of the compatibles and the IBM PC Portable offer the color/graphics-driven display as standard, instead of the monochrome.

The two graphics adaptors can be used to drive a high-resolution IRGB monitor, a black-and-white monitor, or a black-and-white or color television (for those with economy in mind). The enhanced graphics adaptor can also be used to drive the IBM Monochrome display. An IRGB monitor has separate control lines for each of the colors red, green, and blue, and it responds to an intensity modifier. The boards will also drive an RGB monitor, but the extra colors made possible by the intensity signal will not be available.

The color/graphics adaptors have a variety of different output options, which between them dictate the appearance of characters on the display and the type of display hardware that the board is addressing. The enhanced graphics adaptor has a much greater variety of options, including more colors, higher resolution, and downloadable character fonts. It comes with an on-board extension to ROM BIOS that allows access to these functions.

The graphics adaptors operate in one of two modes: text, alias alphanumeric (A/N) mode, and graphics, alias all-points-addressable (APA) mode. (Note: There is opportunity for confusion when using the word *mode*. If you have looked at the original ROM BIOS mode-setting function in an IBM PC compatible you may have noticed that it accepts as an argument one of seven or more modes. To integrate that idea of seven modes with the two we are discussing here, consider the following: the first four of the ROM BIOS modes all belong to the alphanumeric mode; the other three belong to the all-points-addressable mode.) Let's look at these two modes in turn.

The Alphanumeric Mode (A/N) of the Color/Graphics Adaptor

This is the text mode. In it, all 256 characters in the 8-bit ASCII set are representable on the screen (as is true with the monochrome adaptor). The color/graphics adaptor contains a ROM character generator that creates about the same set of characters as are available with the monochrome adaptor, except that they are formed from a 5 × 7 matrix in an 8 × 8 box. The enhanced graphics adaptor can give characters formed from a 7 × 9 matrix in an 8 × 14 box.

In A/N mode the adaptor can create either a 40-character-per-line image or an 80-character-per-line image. In both cases there are 25 lines per screen. The 40-character mode, which doubles the width of the character (but does not change the height), is useful for low-resolution monitors and necessary for almost all televisions.

The memory for the characters that are displayed on the screen is contained on the adaptor. On the color/graphics adaptor 16K bytes of memory are available in all, since in the high-resolution graphics mode 16,000 bytes are required to represent all points on the screen. The alphanumeric mode does not need, but does take advantage of, all this memory. It divides the 16K bytes into eight pages of 40 × 25 screens, or four pages of 80 × 25 screens. In making sense of that allocation, it helps to remember that each character requires an attribute byte to describe how it should be displayed, so an 80 × 25 image requires 4000 bytes. The enhanced graphics adaptor may contain up to 256K of memory. This memory accommodates the extra resolution and colors available with this adaptor.

The Graphics or All-Points-Addressable (APA) Mode

As the name implies, in this mode any point on the screen is directly addressable. The number of points on the screen is determined by the resolution chosen. On the color/graphics adaptor three resolutions are offered. In IBM terminology (which is not universally adopted) they are low, medium, and high, which are 160 × 100, 320 × 200, and 640 × 200, respectively. The 100 or 200 figures refer to the vertical lines of resolution. In the enhanced graphics adaptor two additional resolutions are offered: 320 × 350 and 640 × 350. In the higher resolution modes it is possible to produce quite exciting three-dimensional representations of multiple wave forms over time, to mention just one of hundreds of possible applications.

Character Output in APA (Graphics) Mode

You may sometimes want to annotate a graphic image with text. Unlike the A/N mode, the color/graphics adaptor offers no character-generator

hardware in graphics mode; any characters output to the screen have to be made up by specifying the individual dots. This has the advantage that you can make up your own character sets (which offers much opportunity for creativity); and you aren't even limited to the 8 × 8 matrix.

The disadvantage is that making up your own character set is a lot of work. However, do not despair of character output in graphics mode. If you use ROM BIOS to address the screen in APA mode on the color/graphics adaptor, some character generation is done for you.

ROM BIOS has defined within it the dot patterns for the first normal-sized 128 ASCII characters. (Note: You can look at those definitions in the character-generation table in the ROM BIOS listing found in the *IBM Technical Reference*.) ROM BIOS also includes the routines to send them to the screen dot by dot.

You can create your own set of 128 characters that occupy the upper half of the 8-bit ASCII code (the ASCII characters between 128 and 255 inclusive). ROM BIOS will output them for you if vector 1FH, bytes 0:7C to 0:7F, points to your table of dot positions.

The enhanced graphics adaptor will accept 256-character fonts downloaded to it, for both A/N and APA modes.

Let's now see what color choices are available with the IBM color/graphics adaptor in its various modes. Color selections available with other graphics adaptors (e.g., the enhanced graphics adaptor) will usually match these capabilities and then offer more.

COLOR SELECTION WITH THE COLOR/GRAPHICS ADAPTOR

This is where we introduce you to the color choices and the slightly odd color-selection methods offered by the color/graphics adaptor. A brief summary of the available colors is followed by the specifics of color selection when the adaptor board is operating in A/N mode. The section that follows, on what can be done with color in the APA (graphics) mode, also discusses the memory space on the adaptor board and how that affects the bits available for color choice at the different resolutions. The section on color ends with a summary of the color options available in the medium- and high-resolution graphics modes.

Colors Available with the Color/Graphics Adaptor

(Note: In the text that follows, white, black, and gray are considered colors.)

Sixteen colors are created by the adaptor from three basic colors. These basic colors are the adaptor's versions of red, green, and blue. The 2 × 2

× 2 possible combinations of the three basic colors produce eight colors, including black and white. Each of the eight colors can be modified by the high-intensity attribute, to give the set of sixteen colors shown in Figure 15.1. In this figure, the red, green, blue, and intensity components appear alongside the resulting color.

Let's see how these colors are used in the text (A/N) mode.

Color Choices in the Text, or Alphanumeric, Mode

The color of a character and the color of its individual background within the 8 × 8 matrix are both defined by the attribute byte associated with the character. (Note: If you wish to read more about the attribute byte, see Chapter 5, "Output to the Display Monitors.") In the text mode, color selection offers much scope for experimentation.

The character itself may be any of sixteen colors, and the background any of eight colors. The attribute byte devotes four bits, one each for red,

I	R	G	B	Color
0	0	0	0	Black
0	0	0	1	Blue
0	0	1	0	Green
0	0	1	1	Cyan
0	1	0	0	Red
0	1	0	1	Magenta
0	1	1	0	Brown
0	1	1	1	Light gray
1	0	0	0	Dark gray
1	0	0	1	Light blue
1	0	1	0	Light green
1	0	1	1	Light cyan
1	1	0	0	Light red
1	1	0	1	Light magenta
1	1	1	0	Yellow
1	1	1	1	White

Figure 15.1: *The Color/Graphics Adaptor's Sixteen Colors*

green, blue, and intensity, to the foreground color, hence the choice of sixteen colors. The background color is allocated another three bits, again one each for red, green, and blue, but no intensity bit. Hence, there are eight possible background colors. Filling the attribute byte out to eight bits, the high-order bit makes the character blink (every two-thirds of a second) if set. Figure 15.2 shows the coding of the attribute byte for color selection in alphanumeric mode.

As you can see, a character can be displayed in any combination of eight background and sixteen foreground colors—that is, 128 possible colors. Some of those combinations will not produce a very legible character, especially where the foreground and background colors are the same.

Color Choices in APA (Graphics) Mode

We need to talk about points on the screen, so we'll take a slight digression here into the name of the point on the screen at which the color is applied. This point is variously called a dot, pixel, or pel. *Pixel* is derived from picture element; *pel* is a contraction of it. Why *pel* is a better choice than *dot* is made clear in the description of medium-resolution graphics. For now we will use whichever name seems most appropriate.

Color Choices and Graphics Resolution

The color choices in graphics mode can be related to the number of bits used to represent each point on the screen. To figure out the number of bits per pel available for color specification, we should first talk about how many pels there are in a full screen, and how much memory there is on an adaptor to hold a screen image.

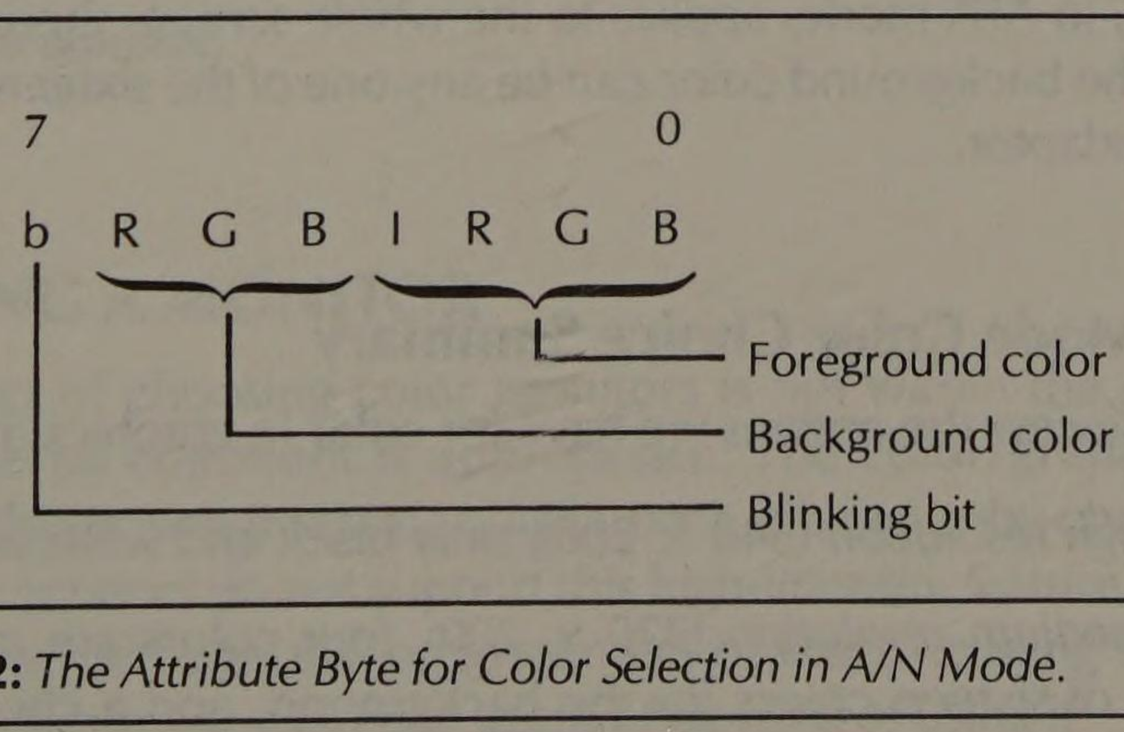


Figure 15.2: *The Attribute Byte for Color Selection in A/N Mode.*

Color Choices in High Resolution In high-resolution mode, the adaptor can address

$$80 \times 25 \times 8 \times 8 = 640 \times 200 = 128{,}000$$

points; that is, 80 columns in each of 25 lines (rows), with each row/column position being filled by an 8×8 rectangle of dots.

With 128,000 bits, at 8 bits per byte, it takes a little less than 16K bytes to store one screen image. The color/graphics adaptor contains 16K bytes of RAM, so it can support only one bit per dot. This allows no spare bits to define the color of the dot, so high-resolution graphics is in black and white only.

Color Choices in Medium Resolution Given the 16K memory on the adaptor, medium resolution (320×200) can use two bits per pel. These two bits can assume four different values. A value of 0 makes the pel assume the background color; that is, it is the equivalent of off. The values 1, 2, and 3 each represent a different color.

So far with medium resolution, we have seen that three colors, plus background, are available for the dot. Now the question is, which three of the sixteen colors do the values 1, 2, and 3 represent? IBM has decided that for us; it has given us a choice of two sets of three. They are as follows:

Set one: Cyan, magenta, and white
Set two: Green, red, and brown

The designers of the adaptor have extended the color capabilities beyond the three colors for the dot (apart from the background color) by including an additional 1-bit color-set switch. This bit dictates which of two color sets the three colors will be chosen from. It applies to the three (nonbackground) colors for the whole screen. The two color sets are defined by the adaptor and cannot be changed.

You may be wondering about the possible choices for the background color, which, in APA mode, applies to the whole screen; there is wild freedom here. The background color can be any one of the sixteen colors available on the adaptor.

Graphics Mode Color Choice Summary

Let's now review the options we have for color in graphics mode.

- *In high resolution* (640×200), only black and white are available.
- *In medium resolution* (320×200), four colors are available: any one of sixteen colors for the background, and a choice between

two sets, each of three possible colors, for the foreground pels; *set one* is cyan, magenta, and white; *set two* is green, red, and brown.

The process for selection of color in medium-resolution graphics mode goes as follows:

1. Choose the background color, any one of sixteen.
2. Choose one of the two predefined three-color sets for the foreground colors.
3. Choose each pel's color from the selected three-color set.

The Best of Pels, Points, Dots, and Pixels

Earlier in this chapter we promised to explain why *pel* is a better word to use than *dot* or *point*. Medium resolution is obtained by this adaptor board by drawing a line, left to right, between even- and odd-numbered high-resolution points. So in medium resolution, what appears on the screen is not a single point—it is a small horizontal line, more naturally called a picture element, or pel.

A SMALL PUZZLE FOR SOFTWARE WRITERS

Those of you familiar with the contents of the attribute byte may be wondering what happens when a character attribute originally designed for the monochrome display is sent out via the color/graphics adaptor. What will the character look like?

Reverse video still works; however, it affects only black-and-white characters. The attribute-bit setting for the underline feature in monochrome results in a blue character when processed by the color/graphics adaptor. The reverse translation takes place when blue characters are output via the monochrome adaptor.

CHOOSING A MONITOR

The subject of choosing color monitors is not within the scope of this book, but some comment is appropriate. The color/graphics adaptor allows specifying high-intensity output on a character-by-character basis. Some color monitors do not support this high-intensity feature. You can live without it, even though the programmers of much existing software (the WordStar program, for example) seem to be fond of using it.

Mixing and Matching

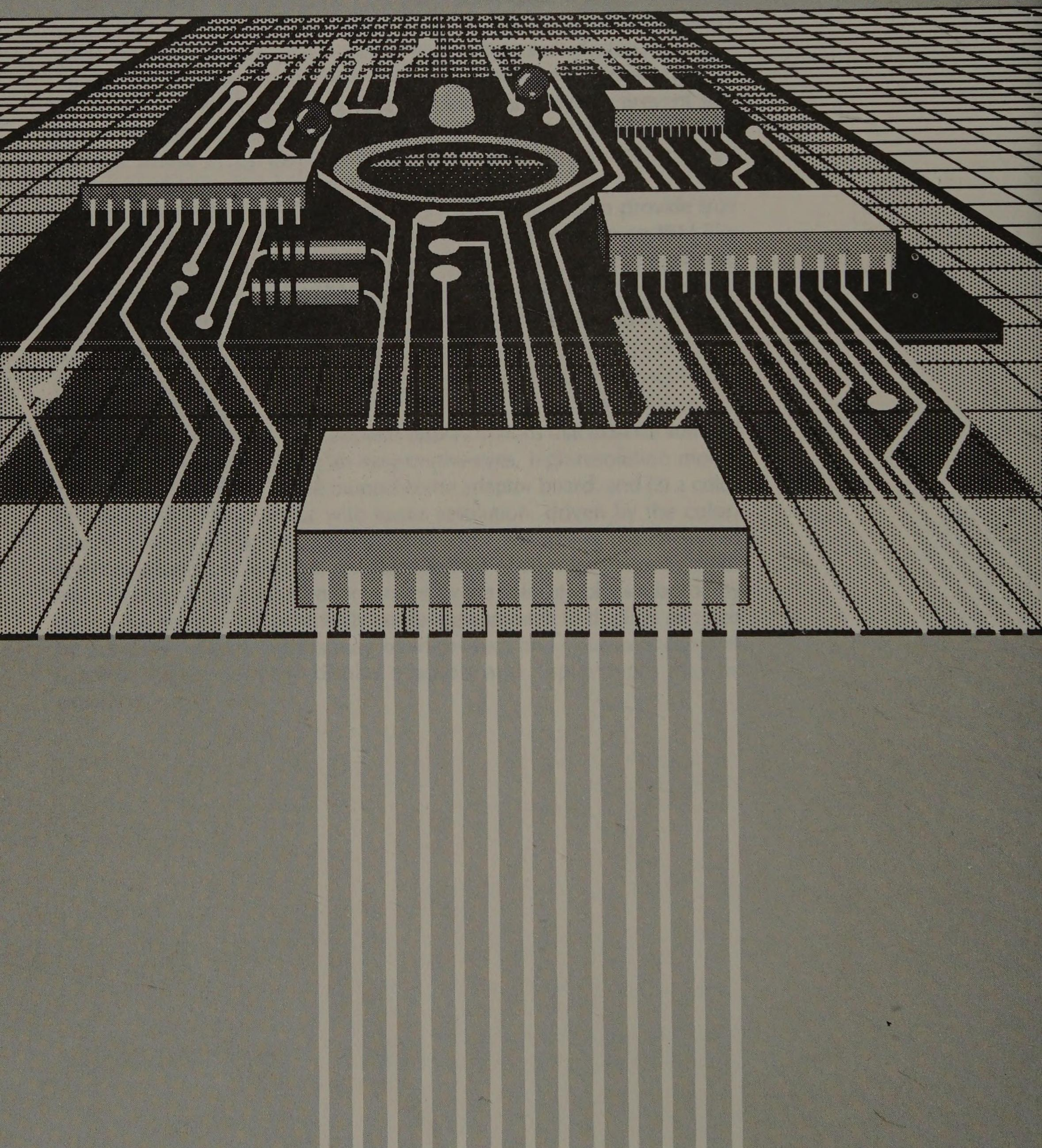
You might wish to try using the color/graphics adaptor to drive an IBM-like monochrome monitor, or the monochrome adaptor to drive an ordinary black-and-white monitor. Don't casually mix monitors and adaptors.

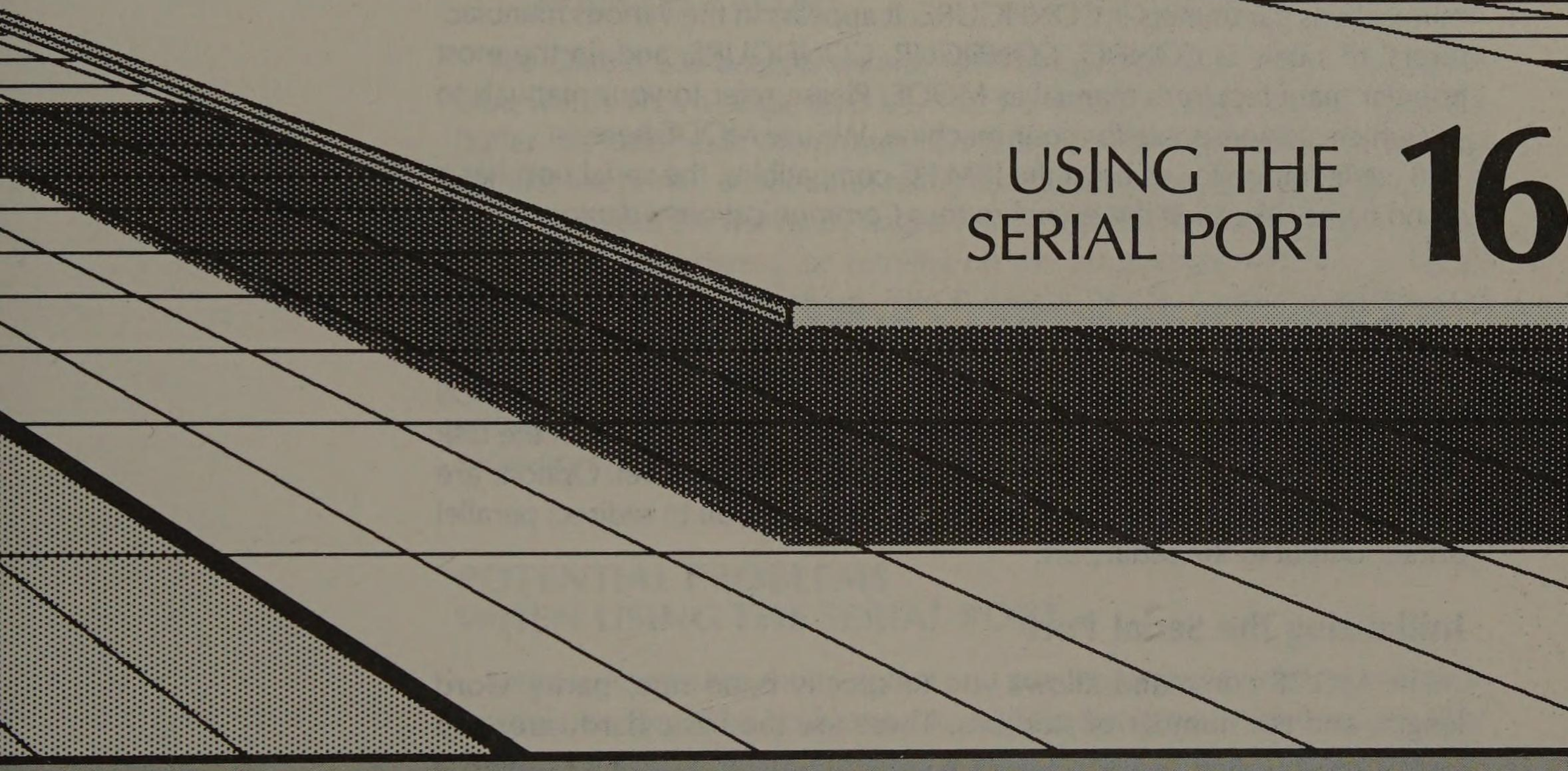
There are important technical differences between the electrical output of the color/graphics and monochrome adaptor boards, which prevent successful long-term driving of the monochrome screen by the color/graphics adaptor. Smoke accompanying monochrome monitor damage has been reported as the result of one such attempt.

Some adaptor boards offered by other manufacturers can provide true graphics capabilities (not just block-character graphics) on an IBM-like monochrome display unit. These are presumably safe to use. In general, the display must be matched to the adaptor driving it. If you buy through mail-order houses, make sure you will receive compatible components.

SUMMARY

We have seen the two most popular display options that exist for some of the IBM PC compatibles: (1) an easy-on-the-eyes, high-resolution monochrome display, driven by the monochrome adaptor board, and (2) a color or black-and-white monitor with lower resolution, driven by the color/graphics adaptor. We have briefly noted some features of the enhanced graphics adaptor. In addition, we discussed the options for choosing color and creating graphics on monitors driven by the color/graphics adaptor. By now, your grasp of the potentially confusing approach to displays adopted by computer manufacturers may be improved, and your awareness of some of the variations in display adaptors and display types may be extended.





USING THE SERIAL PORT 16

INTRODUCTION

This chapter shows how to use the serial port that may be installed in your computer. The serial port is your link with the world beyond your desk top. With a serial port, a modem, the right software, and a telephone line, you can gather and exchange information with bulletin boards, data banks, perhaps your own bank, Western Union, assorted vendors, and other computer users. Used in a more mundane way, the serial port allows you to use printers that have only serial interfaces, by redirecting output destined for the parallel port to the serial port.

We will begin the chapter with a brief discussion of the communications capabilities within MS-DOS at the user level. We will give an example of how to use these capabilities to run a printer with a serial interface, continuing with a note on how success is not always hard to obtain. We will discuss the RS-232 "standard," and cover details of cabling that may be relevant, including what is meant by a "null modem."

The generic name of the user-level command that allows control of communications parameters is CONFIGURE. It appears in the various manufacturers' manuals as CONFIG, CONFIGUR, CONFIGURE, and, in the most popular manufacturer's manual as MODE. Please refer to your manuals to see which is appropriate for your machine. We use MODE here.

In the literature for some of the IBM PC compatibles, the serial port has a grand name; it's called the *Asynchronous Communications Adaptor.*

MS-DOS COMMUNICATIONS CAPABILITIES

At the user level, MS-DOS can do a lot for you. The authors of MS-DOS have provided a MODE command that is an essential ingredient in the task of moving files of data across serial ports at the command level. Options are provided that initialize the serial port, and that allow you to redirect parallel printer output to the serial port.

Initializing the Serial Port

The MODE command allows you to specify baud rate, parity, word length, and the number of stop bits. These are the basic hardware- and software-controlled factors that vary according to the device doing the communication and the software driving it.

The trick to the successful use of this command is to know what to tell it. More on that later. The DOS manuals are clear about the options you have and give useful examples.

Redirecting Parallel Printer Output to the Serial Port

The MODE command allows you to use printers with serial interfaces. It tells MS-DOS to switch the output destined for (one of) the parallel printer port(s) to a serial port. Again, the DOS manuals are clear on how to use the command. Let's look at an example of printer output redirection.

USING A PRINTER WITH A SERIAL INTERFACE

We will look at the specific MODE commands for using a printer with a serial interface. The printer is the Diablo 630.

Output to the Diablo 630

The commands to achieve this redirection (output to the Diablo 630) are as follows:

```
MODE COM1:1200,N,8,1,P
```

and then:

```
MODE LPT1:=COM1
```

The Diablo 630 accepts data at 1200 baud, uses no parity, and accepts 8-bit words with a single stop bit. The P is necessary because the printer's buffer fills faster than the printer can print the characters. When that state occurs, the printer sends a not-ready condition to the serial port. When the serial port sees the not-ready flag, it has the option of abandoning the communication in progress, or retrying on the assumption that the not-ready condition will soon clear up. The P option tells the software to hang in there, even if it gets a not-ready condition.

If all this seems quite easy, you may be correct. A number of things can go wrong with this process, however, and at the risk of turning readers into serial-port hypochondriacs, here is a discussion of some of them.

POTENTIAL PROBLEMS WHEN USING THE SERIAL PORT

Let's assume for the moment that you are attempting to use a printer with a serial interface. The problems you might experience involve one or all of the following: cabling, lack of information about the printer characteristics, and, sometimes, software problems.

Cabling, or What Do These Guys Think RS-232 Means?

The serial port uses the RS-232 standard. So do many printers. "Good," we conclude, "the two devices can talk to one another." That conclusion is not valid. In my experience with Silicon Valley manufacturers when a new device needs to be connected to the outside world, the engineers think "Well, how shall we implement the RS-232 standard this week?" Their problem (and ours) is that the RS-232 standard has to deal with a wide range of communication complexities. For connecting simple devices (say, truly dumb terminals), only a couple of wires are necessary. The more complex the communication requirements, the greater the number of signal paths required. That means more expense for the builder and the customer, both for the more bulky cable and for the more complex devices required to drive the communications.

The RS-232 standard has history working against it. When terminals were so dumb they didn't care—they could be hooked up with just two wires, so the RS-232 standard was widely ignored—it dealt with communications complexities far greater than were necessary at the time. Programming is just crawling out of the primeval slime of everyone doing it the way they think (know?) will work OK, and there are indications that engineers who design the communications capabilities of a device are getting used to the cheapness and reliability of greater device intelligence and are planning for it. In the meantime, in order to figure out how to connect a printer to your computer's serial port, you will only need:

- engineering drawings of the printer interface and the serial board;
- a complete knowledge of the software operating the devices at each end;
- an oscilloscope and the ability to use it;
- cable, connector parts, and a soldering iron;
- imagination and patience;
- and, lastly, lots of experience.

Lack of Information about Printer Characteristics

This has long been a problem, but there is hope. The diatribe above ignores the good work that is being done these days by some printer manufacturers. A few printers are shipped with adequate information on how to make them work with specific computers. For users of the IBM PC compatibles, there is even more hope. IBM has open specifications on the PC, so printer manufacturers can build to suit either the PC's asynchronous or parallel communications. Printer manufacturers may also provide documentation showing what baud rate, parity, word length, and number of stop bits to use.

Cabling the Serial Port

The number of connections between the serial port and the device it will drive will vary according to the needs of the device and the sophistication of the software driving the device. By far the simplest approach to successful cabling is to find another user with the same computer who has successfully done what you want to do, exactly matching both software and hardware. Once you have found such a user, ask him or her who supplied the cable, and to print the command sequences used to get the device going.

It may be that the experience of arriving at the correct cabling and command sequence solution was painful. In that case, you may detect a slight

hesitation or unwillingness to part with such dearly bought information, just for the asking. Offer them a weekend in Reno or Atlantic City. This shows them how much you value the information they have to offer.

If you can't find a cable supplier who can satisfy you that the cable you are buying will work for the exact situation you have in mind, and you have found no one who has done what you want to do, check the documentation that is supplied with the device you are connecting to the serial port. Look for pin connection diagrams. As a last resort, call the manufacturer.

In this chapter only a brief outline of the essentials can be provided. There is an excellent book all about this subject: *The RS-232 Solution* by Joe Campbell (SYBEX,1984). In it you will find the solution to such mysteries as using an NEC Spinwriter 3510 with an IBM PC, and connecting your computer with modems.

A Null Modem

If you want two MS-DOS computers to talk to each other, and you don't want to use modems and the telephone to connect them, you will need a null modem. A *null modem* is a device that connects one computer's transmit data line (pin 2 on its serial port) to the other computer's receive data line (pin 3 on the other computers's serial port). Another matching pair of lines are data terminal ready (pin 20) and data set ready (pin 6). Request to send (pin 4) and clear to send (pin 5) need special handling. Figure 16.1 shows a set of pin connections that serves as a null modem.

Pins 4 and 5 are jumpered together at one end, so both connect with pin 8 of the other serial port. (In Figure 16.1, lines that cross in the central part of the figure are not electrically connected.) As usual in the world of RS-232, you will find that this is not the only form a null modem can take.

CONNECTION WITH INFORMATION RETRIEVAL SERVICES

To connect with the outside world, you will need a modem, the serial port, a cable to connect the two, a telephone, and some communications software. We will assume that you have all the hardware. The hardware manufacturer will often supply the communications at a discounted price if you buy it with the hardware. For example, Zoom Telephonics makes pfs: Access available inexpensively.

Communications Software

A variety of commercial software is available. Crosstalk is one good possibility, as is the latest version of PC-TALK. The latter is very attractive because

of its quality as well as its price ($45). IBM sells the Asynchronous Communications Support (2.0) package, which includes a manual as well as asynchronous communications software.

The function of these programs is to establish the communications protocol appropriate to the information service being accessed, and then to transfer the characters typed at the keyboard to the serial port and send characters received from the serial port to the screen and/or to a file.

Well-written communications software will normally keep at a distance all the messy details of communications such as baud rate, number of stop bits, half- or full-duplex operation, etc. There may be times when you will need to know about these details—well-written communications software usually has a mode where you can specify parameters. You may also use the MODE command to initialize the protocol parameters.

SUMMARY

We have learned how to use the MODE command to redirect output to a serial printer. We have seen how the physical connection between the serial port and the external device may not always be easy to establish, and how another person's successful experience is valuable. We have seen that

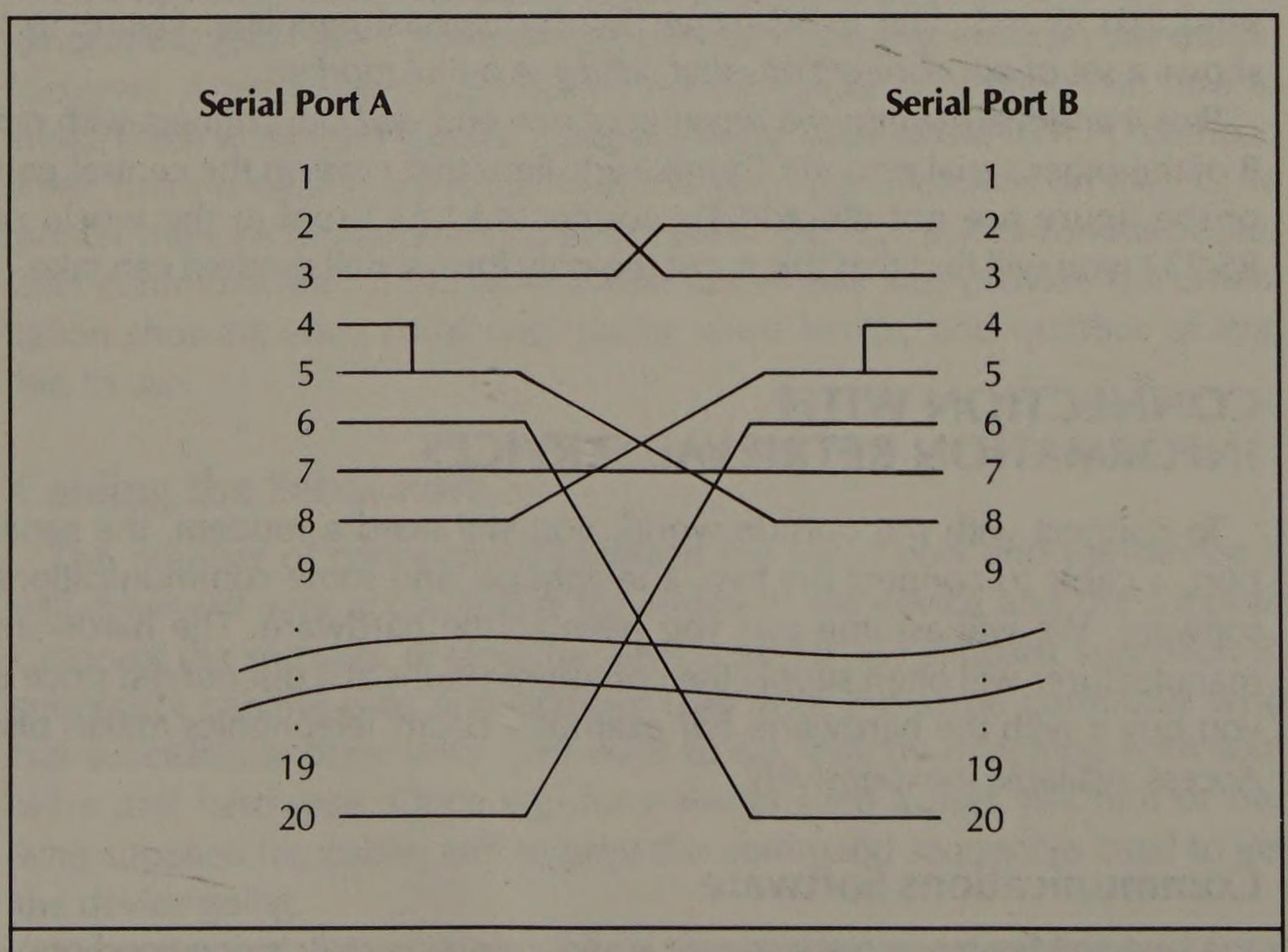
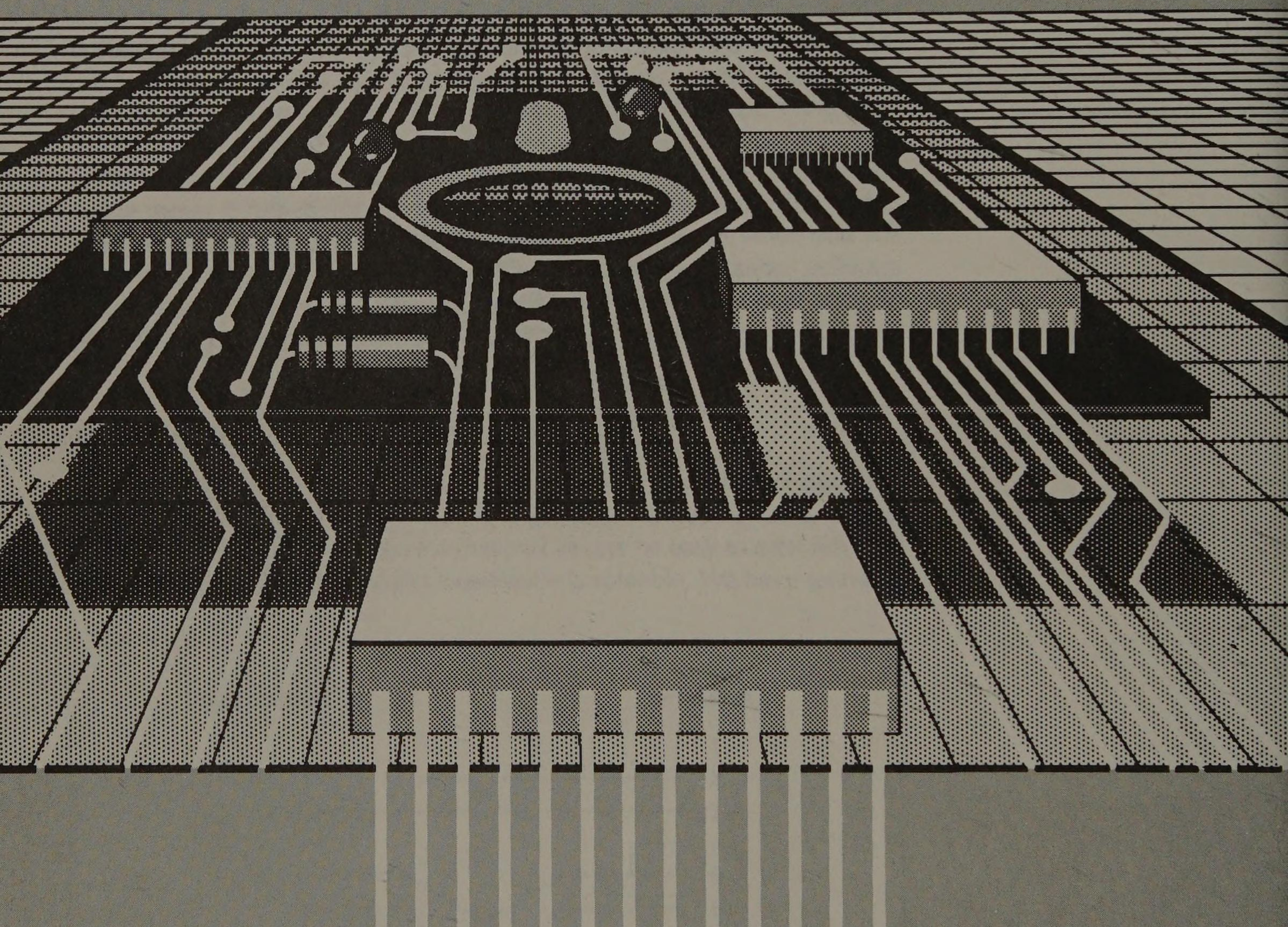
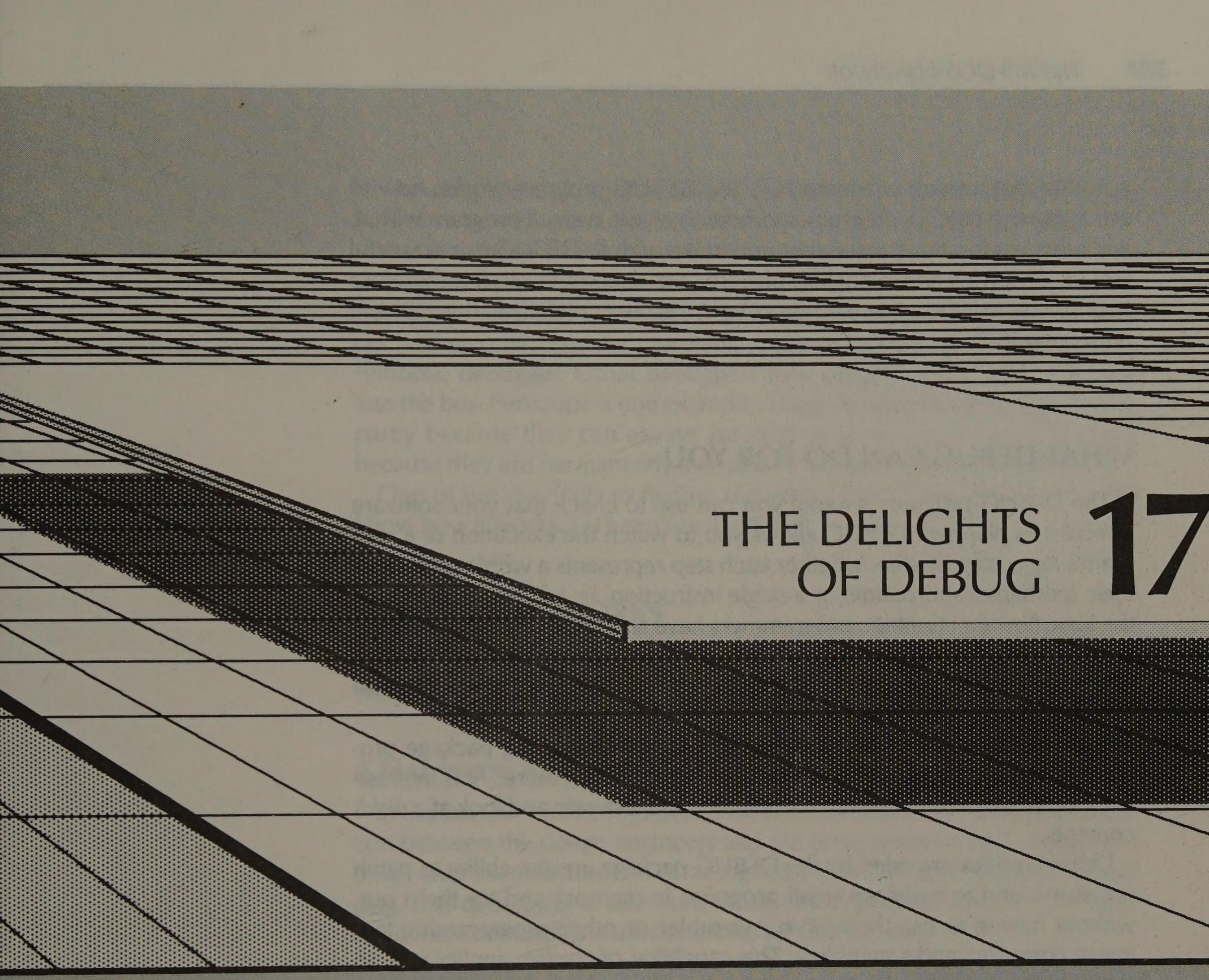


Figure 16.1: *Null Modem Pin Connections*

communications software is necessary to connect with the outside world, and that such software is available from a variety of sources.

Now we'll change directions completely. We will forget the outside world and look at DEBUG, a program that lets us look into the innards of MS-DOS.





THE DELIGHTS OF DEBUG 17

INTRODUCTION

DEBUG is your window into the workings and wonders of MS-DOS. It also has the less spectacular but equally powerful function of helping you find bugs in your programs and patch them. This chapter is directed to those who have had little experience with debuggers; that is, it is an introduction to debugging, containing little for the experienced assembly-language programmer.

In this chapter you will learn how the DEBUG program works, how to use it, how to patch a program and how to create a small program with it, and what explorations you might undertake with it. DEBUG is a powerful tool for reviewing and changing software at its deepest levels; it is therefore appropriate to issue a few warnings about treading carefully in the software undergrowth.

WHAT DEBUG CAN DO FOR YOU

The DEBUG program is a tool you can use to check that your software operates as planned. DEBUG allows you to watch the execution of a program's logic, step by step, whether each step represents a whole module of logic (perhaps a subroutine) or a single instruction. In addition to watching the logic flow (or stumble), programmers have an intense interest in before-and-after views of code and data. Both of these move around in memory, and from disk to memory and back. DEBUG enables you to look at these before-and-after states in detail.

You can see that one of the major functions a debugging package provides is that it gives you a chance to "see what is really there." It is interesting, as well as useful, to poke around inside the software and look at a disk's contents.

Other facilities provided by the DEBUG package are the ability to patch programs, and to make up small programs in memory and try them out, without having to use the Macro Assembler or other programming language compilers and assemblers. This capability, of course, applies only to small sections of software. A sequence of anything above 50 to 100 instructions is probably stretching it. "Creating Programs with DEBUG," later in this chapter, shows you how to create short programs and save them on disk.

To make full use of DEBUG, you need access to descriptions of the assembly-language instructions, and you must be willing to learn something about assembly-language programming, if you don't know about it already. You also need to know something of the architecture of the program being debugged and the hardware it is running on. The first thing to understand is how the registers in the 8086/8088 are assigned and used (see Chapter 1).

DEBUG is a powerful tool, and it works best in the hands of an experienced and knowledgeable user. If this is your first experience with a debugging package, don't be surprised if your questions regarding how a certain piece of software works are not answered within the first half-hour of using the debugger. You will likely be frustrated by strange and apparently inexplicable circumstances. That's when you either have to learn more or skip that approach and try another. If you don't know much about how computers

work, using DEBUG is a little like being tossed into the deep end to learn how to swim—for some unpleasant, for others (terminally) ineffective.

You may have heard of *symbolic* debuggers. These more advanced debuggers have access to the symbol table(s) of the program(s) being debugged. With them, you can debug high-level language programs without having to dig deep into assembly language. However, DEBUG is not a symbolic debugger. Other debuggers exist on expansion cards that plug into the bus. Periscope is one example. These debuggers are very powerful, partly because they can always get control of the processor, and partly because they are permanently available to examine any odd occurrence.

One of the shortcuts to finding out what DEBUG can do for you is to know how it works. Let's explore that topic.

HOW DEBUG WORKS

DEBUG uses debugging-oriented hardware features of the 8086/8088 as well as software to do its job. Let's look over the hardware features first.

A Hardware Assist for DEBUG

In the last few years computers have been designed with better cooperation between the design engineers and the programmers. Intel, along with other chip manufacturers, has recognized that software requires debugging, that debugging is time-consuming and expensive, and that if a chip has decent debugging features built in, it will sell more.

DEBUG uses the trace and breakpoint features built into the 8086/8088. (Debuggers for earlier microcomputers worked much harder to achieve the same effect, and they had to overcome a variety of problems that had to do with timing and instruction size. These problems no longer exist, thanks to the design of the 8086/8088.)

The 8086/8088 is designed so that when the next command to be executed is a single byte of CCH, an interrupt occurs, execution of the current path of logic is suspended, and control is transferred to the routine pointed to by interrupt vector number 3. All the DEBUG package has to do to get itself called at a specific instruction is to replace the first (and perhaps only) byte of that instruction with the break instruction (a byte containing CCH), and make sure that vector 3 (absolute locations 0CH – 0FH) points to itself (the debugging package).

Intel also reserves interrupt vector 1 (locations 04H–07H), through which control is transferred at the end of every instruction execution if the CPU is operating in trace mode. The CPU operates in trace mode if the trap flag is set in the program status word (PSW).

So you see, there are two easy ways for the debugging software to get control of any stage of program execution:

1. Inserting breakpoint instructions, which replace the first byte of an instruction with the one-byte break instruction.
2. Initiating trace (alias single-step) mode, by turning on the trap flag in the program status word.

Debugging Read-Only Memory (ROM)

As you might deduce, the first method cannot be used for debugging or stepping through ROM, because the debugger can't write the break instruction into it. To see what happens in ROM, you should use the second method, single-step, to trace your way through it.

Let's look at some of the setup operations that DEBUG has to go through to get ready for debugging an existing program.

The Software Components of DEBUG

DEBUG is an "envelope" around the program being debugged. At the prompt level, when you first call DEBUG, COMMAND.COM loads the debugger. If the name of the program to be debugged has been specified as an argument, DEBUG loads the program defined by the argument into the memory that is left after DEBUG has been loaded. It then correctly initializes the program segment prefix (PSP), which is the area at the beginning of a program that formalizes its connection with MS-DOS. DEBUG sets up interrupt vectors 1 and 3 and after that, the user is given control.

If no arguments have been specified, the user is given control immediately. The program (or data) loading process can be invoked later, by using the Name and Load commands.

Let's look at how to use DEBUG.

HOW TO USE DEBUG

We will begin this section with a brief introduction to the commands within DEBUG. We will then throw in some perhaps gratuitous advice on what should be in your head when you use it, and then review, again briefly, the commands and facilities worth special note.

A Brief Summary of DEBUG's Capabilities

Figure 17.1 shows in the left-hand column the things you might want to do with DEBUG; the right-hand column lists the commands available.

Commands can often be put to good use for purposes other than those shown here.

A more complete summary of the DEBUG commands can be found at the end of this chapter.

The Theory and Morals of Debugging

The best debugging tool, if you wrote the program or have well-documented, accurate listings, is your head. A lot of debugging time can be saved if you first think carefully and clearly about the potential sources of the problem. You may be one of those people who tend to get sidetracked by interesting circumstances when digging around inside a program with the debugger. If you are, prepare a well-thought-out search plan first; it will save you a lot of time. In other words, use your powers of logic and visualization before resorting to the debugger.

However, that recommendation is not always easy to follow. If you want a hero in this field, there is Nikola Tesla, the inventor and engineer who patented the method for generating alternating current. Among many startling accomplishments, he possessed an ability to visualize inventions with great clarity and intensity. Prior to registering his AC power generation and distribution patents, he designed, assembled, ran, disassembled, and finally examined for wear an AC electric motor—all in his head. A few rare programmers can do the same sort of thing with software. Give it a try!

Function	Command(s)
Program execution control	Go, Trace, and Proceed
Disk access	Name, Load, and Write
Data display	Dump
Code display	Unassemble
Data entry	Enter and Fill
Code entry	Assemble and Enter
Status display and entry	Register
Single-character I/O	Input and Output
Search for byte(s)	Search
Compare memory	Compare
Mental aids	Hexarithmetic and Assemble

Figure 17.1: *DEBUG's Command Capabilities*

Now that we have covered the homily of "think about the problem first; use the debugger later," let's return and look at the debugger we are going to need.

Practical Debugging

The first step in any debugging venture is to isolate the problem, that is, find out when, during the execution of the program, the problem occurs. Doing this will narrow the search to a limited portion of the code. The next step is to see what makes the bug occur—at which instruction in the isolated group of instructions the performance is not as expected. This two-step debugging approach roughly corresponds to the difference between using the Go command and using the Proceed and Trace commands in DEBUG.

Go is a good tool for checking your program logic at the modular level. In contrast to the Trace command, it is your coarse-adjustment tool. With it you can follow the program in major leaps until it goes wrong, then back off and examine the last step in smaller increments. At the risk of overstressing a point, when you have narrowed down somewhat the source of the logic failure, take the time to get your head away from DEBUG and think about the program. The bug will often be clear to you long before you can pinpoint it exactly with the Go, Proceed, and Trace commands.

If the bug doesn't yield to the Go-with-breakpoints approach (breakpoints are program locations containing the break instruction CCH), you may have to resort to the Trace and Proceed commands. These commands are useful when a piece of code doesn't do what you think the listing dictates.

You need commands to look at the data before the bug occurs, and at the wreckage afterwards. For these purposes, DEBUG provides the Dump, Register, and Compare commands.

Let's look at a few of DEBUG's commands, and see how each one can be used.

DEBUG's Commands Reviewed

The DOS manuals give good descriptions of each of the commands in DEBUG. The manuals also have sections on information common to all commands, as well as definitions of the command parameters. There you can find pretty much all you need to know about DEBUG's capabilities. We recommend you read those descriptions; they show you features of DEBUG that are not always available in other debuggers, and that are a

pleasure to find. Here are a couple of examples of these features:

1. The ability to mix data in several different formats in the same command line when entering data using the Enter command:

   ```
   E DS:100 F3 "abc" 0D
   ```

   Starting at DS:100, the five bytes F3H, 61H, 62H, 63H, and 0DH are entered.
2. The Dump and Unassemble commands keep track of where they were last used, and continue from that point if no arguments are given the next time they are invoked.

The Go Command

The Go command starts or continues execution of the program you are debugging (unless you give it a new address at which to begin execution by using the G=address option). Breakpoint instructions are inserted at any addresses you specify. A typical Go command to start the execution of a small program and trap it at one of three places would be

```
g=100 10c 116 134
```

Note that the CS register was not specified; the current value is assumed. If the break occurred at 10C, the next Go command might be

```
g 134 147
```

This continues execution from the 10C breakpoint. Note the need to respecify the 134 breakpoint. (The example assumes that the 116 breakpoint is no longer relevant; the 10C breakpoint is on a different logic path.)

If you think that at least one of your breakpoints will not be reached, add the address of the instruction prior to the program's termination as another breakpoint.

All breakpoints are removed when one of them is reached. This wholesale removal of all breakpoints can be annoying in some circumstances. If you want a permanent (i.e., for the duration of the debugging) breakpoint, you can set it by using the Enter command to place a CCH at the breakpoint location. Remember that DEBUG won't remove this permanent breakpoint for you.

In general, run your program with BREAK OFF. If BREAK is ON, that logic gets interwoven with DEBUG's in a confusing way.

The Trace Command

Trace executes one instruction at a time, before handing control back to you, unless it is given an argument that tells it how many instructions to

execute. You will always get a display of the registers and flags after the execution of each instruction, so it is a very slow way of stepping through a program. The command to trace the next five instructions is

```
t 5
```

The Proceed Command

The Proceed command is a more powerful version of the Trace command. Proceed single-steps through a program, but it skips examination of the details of subroutine and interrupt calls, and you don't see each iteration of string and loop instructions.

The shortcuts offered by Proceed can be accomplished fairly simply by using the Go command, but using Go requires you to specify the address of the instruction following, say, the subroutine call. With Proceed, you don't have to think that hard.

The Write Command

The power of devastation is in your hands when you use the Write command. If you are doing anything out of the ordinary, write onto a *copy* of a disk, never onto an original from which no copies have been made.

The Assemble Command

With the Assemble command, you can write small routines in assembly language. If you are not going to do anything but minor work in assembler, the existence of this command can save you having to learn how to use the Macro Assembler. To add instructions at, say, location CS:134, type

```
a134<return>
```

and then type in the assembly-language instructions.

The Fill Command

Fill is a handy command, but occasionally devastating if mistyped. A command to zero all the bytes of a buffer into which you are about to read one sector of a disk is

```
F 400 L200 0
```

where DS:400 is the buffer address.

The Register Command

When you are debugging a program, just after it has been loaded (via the Load and Name commands or as an argument to the DEBUG call), the registers are all set to begin execution of the program. Thus, you can use the Register command to find where the program is loaded in memory.

We have not said much about the Name, Load, and Write commands; let's look at a procedure for patching a program, which is one place where those commands are likely to be used.

PATCHING WITH DEBUG

When you patch a program, you change the contents of one or a few of its locations to alter the way the program operates. Patching with DEBUG is quite simple. You need to remember that the program begins at CS:100, not code segment address zero (CS:0000, where the program segment prefix is found). Here are the steps for patching, say, location 273H:

1. Call DEBUG PROGNAME.COM, where PROGNAME.COM is the name of the program you wish to patch.
2. Use Dump or Unassemble to look at location 373H. (Remember, your program begins after the 100H-long PSP.)
3. Use Enter to make the change.
4. Use Dump or Unassemble again to check it.
5. Use Write to write the patched version onto the disk.

Let's look at a more specific, but simple, example. You have written, in assembly language, a program that first announces itself, then does a variety of small tasks, and then displays all the ASCII characters above 127. The program is very short and the display-loop portion of it looks like this:

```
006A B402       DISP MOV AH,2       ;Display output function
006C BA7F00          MOV DX,7FH     ;Set to start at ASCII 128
006F 42         INCH INC  DX        ;Bump to next character
0070 CD21            INT  21H       ;DOS function call
0072 81FAFF00        CMP DX, FFH    ;Done yet?
0076 7EF7            JLE  INCH      ;Loop if not
```

Let's assume you want to change the program to display all the ASCII characters, not just the last 128. This can be done by changing the word at location 6D to minus one (−1). If you look at the second line of the display program, the word at 6D is the value of the first character (before it is incremented by one) to be passed to the display output function.

The name of your program is DISASCII.COM. Let's call the debugger. (Note: In the following examples, your manufacturer's version of MS-DOS may use different prompt characters; for example, you might see A: instead of A>. Similarly, DEBUG may use a different prompt from that shown in the examples. The DEBUG prompt might be : or > instead of –.) To call the

debugger, type the characters following the A> prompt:

```
A>debug disascii.com
```

You will see DEBUG's dash (or other) prompt. To look at the code in the area of interest, type the characters following the dash prompt below:

```
-u 16a
```

Note the addition of 100H to allow for the program segment prefix. You will see perhaps fifteen lines of your program, the first six lines of which look like this:

```
08F1:016A B402          MOV     AH,02
08F1:016C BA7F00        MOV     DX,007F
08F1:016F 42            INC     DX
08F1:0170 CD21          INT     21
08F1:0172 81FAFF00      CMP     DX,00FF
08F1:0176 7EF7          JLE     016F
```

Check that locations 16D and 16E are indeed the ones that need to be set to −1. Once you are sure of this, type

```
e 16d ff ff
```

This changes the word at location 16D to −1. Or, you could enter the Move instruction again with a different value, as follows. Type

```
a 16c
```

You will see

```
08F1:016C
```

which you follow by typing

```
mov dx,ffff
```

or

```
mov, dx, -1
```

To check that you have made the change correctly, type

```
u 16a
```

and look at the first instruction. (Note: if you are using DEBUG to try out this display loop, you could add at the end the following instruction:

```
08F1:0178 EBF0 jmp 16a
```

This extra instruction keeps execution within the bounds of the display program. Without it, careless use of the Go command would send the program into never-never land.)

If the program now looks correct, you are ready to write this patched version onto disk. We assume that you have another copy of the original program stashed away in case something goes wrong (as mentioned before, the chances of writing incorrect data in the wrong place on the disk are fairly high for beginners, and still high for the experienced, even though DEBUG does what it can to help you avoid this disaster). To write the patched version in place of the one we read, type

```
W
```

This writes the changed program out to disk, and the job is done. You can see from this example that DEBUG has been written to be easy to use. All sorts of variations in this patch procedure are possible. Read the DOS manuals for information on how to patch .EXE files, how to use the Name and Load commands to bring in the file to be patched, and how to use arguments in the Write command to write the patched file to another part of the disk.

CREATING PROGRAMS WITH DEBUG

It is easy to create small assembly-language programs using DEBUG. We will discuss the sequence of operations, then go through an example. To use this method of creating short, useful programs, you need to know a little about assembly-language programming, but you don't need to have experience specifically in 8086/8088 assembly language.

We will use the Name and Write commands within DEBUG to create an executable program (a file) with the name PROGRAM.COM. Once the file has been created, typing PROGRAM in response to the MS-DOS prompt will execute your program (the contents of the file). Let's outline the necessary steps:

- Call DEBUG.
- Use the Name command to specify the name of the program file.
- Create the program, starting at location 100H, and check it out.
- Use the Write command to write it to disk.

As you can see, it is not very complex. Even when we examine the steps in detail, we shall see that most of the work is done for us by DEBUG.

The example program that we are going to write will be called PROGRAM.COM. Its function is to display a message and then quit. Not very exciting, but adequate to demonstrate how easy it is to create small

programs without using the Macro Assembler. Let's go through the example.

1. Decide on the name of the program. The extension part of the program's name must be .COM.
2. Make sure the disk on which you want the program written is in one of the disk drives, and that the disk (and directory) has enough room for a program.
3. At the MS-DOS prompt, type

```
debug
```

4. Use the Name command to tell DEBUG the name of the program it is creating. We will put the program on drive B. Type

```
n b:program.com
```

   DEBUG formats the FCB at 5C to match this file name.

5. Create the program. This stage is presented in three parts, 5a through 5c.

5a. In this step we use the Assemble command. Type

```
a100
mov ah,9
mov dx,200
int 21
mov al,0
mov ah,4c
int 21
```

The first three instructions (after the Assemble command—a100) display the string at DS:200; the last three terminate our program with a zero (meaning "all is well") return code. (Note: The program begins at location 100H. This leaves room for the program segment prefix, which MS-DOS constructs in the first 100H bytes of the program's segment.)

5b. To enter the message that the first three instructions will display, type

```
e200 'This is a ' 0a 'short message.$'
```

The 0a value in the middle of the message is a line feed.

5c. Check that all instructions and data are correctly entered by typing

```
u100
```

and

```
d200
```

The Unassemble command displays the instructions, and the Dump command should reveal the message.

6. Check out the program by typing

```
g=100 107
```

This instruction does not check out the termination portion of the program; it just displays the message. If you check out the termination portion of the program, you will find yourself back at the prompt, out of DEBUG. All is not lost when that happens. You can recall DEBUG and use the Name command again. The instructions and data will still be there.

7. Set the BX and CX registers to the length of the program. We know that this is a very short program. It begins at CS:100, and its data ends soon after DS:200. We already know that CS and DS are set to the same value, because DEBUG set them that way when we called it in step 3. We know also that only multiples of whole sectors are written to the disk. Our program length is clearly less than 512 (200H) bytes, so we set CX, the low-order part of the program length, to 200H by typing

```
rcx
200
```

In our example, BX does not need altering, because it was set to zero when DEBUG was called. If your program is at all complex, there is a good chance that during checkout the BX register was used. If so, to set the BX register to the correct length (0 in our case), type

```
rbx
0
```

We don't have to use exact multiples of 200H. In fact, when the program is complete, it is best to be precise about the length of the program.

8. Write the program to disk using the Write command. The length is specified by the CX and BX registers, and the name was specified using the Name command in step 4, so DEBUG knows what to do. Type

```
w
```

DEBUG displays a message that tells you it is writing the number of bytes you specified. No address need be specified—our program begins at 100H, which is the default value that the Write command uses if no address is specified.

9. Quit the DEBUG program by typing

```
q
```

10. When you see the MS-DOS prompt, type

```
b:program
```

You should see

```
This is a
short message.
```

Altering an Existing Program

The procedure for adding to or changing an existing program is very similar. The two differences are (1) when you call DEBUG, you type

```
debug b:program.com
```

and do not have to use the Name command (step 4 above), and (2) after DEBUG is loaded and you see the dash prompt, use the Register command and note the CX and BX values. CX and BX contain the length of the program, and they need to be reset to these values (or expanded if necessary) before you use the Write command to write out the changed (or expanded) program.

EXPLORING WITH DEBUG

DEBUG can be used to look around the programs that make up MS-DOS, and to look around ROM BIOS. With patient exploration, you can learn a lot about programming and operating systems with DEBUG. To begin with, here are a few places of interest that you can look at with DEBUG.

The Interrupt Vectors

The 8086/8088 has the first 1K of memory reserved for interrupt vectors. MS-DOS uses vectors 20H through 2FH. See the section "Interrupt Vectors" in Chapter 1 for more information.

ROM BIOS Communications Area

The ROM BIOS data area begins at location 400H in the IBM PC compatibles. It contains a variety of information, some of which describes the current hardware configuration. Among these values is the equipment flag word at 410H. What else is there is revealed by the ROM BIOS listings found in the *IBM Technical Reference* and in Figure 17.2. The meaning of the bits in the equipment flag for the IBM PC, where the bits are numbered from low to high order, are shown. Take care when interpreting the values for pairs or triplets of bits.

Most programs use the ROM BIOS display output routines, as MS-DOS does. These routines use the equipment flag to indicate whether output should go to the screen via the monochrome or the color/graphics adaptor. If you have both devices, you can switch output back and forth between the two by changing the value of the equipment flag. Doing this is not without its hazards. One thing to remember is that although you may have patched the wrong value in there, and nothing is showing on either screen, keyboard input is handled by a different service routine. If you remember this, faithfully entering the DEBUG keystrokes that reset the screen output to monochrome can bring your system back to conventional life. See the section "Switching Output between Monochrome and Color/Graphics," toward the end of Chapter 5.

Bit(s)	Meaning
0	Set if system has disk drives
1	Not used
3,2	System board RAM size, where 00 = 16K, 01 = 32K, 10 = 48K, 11 = 64K
5,4	Initial video mode, where 01 = 40 × 25 color, 10 = 80 × 25 color, 11 = 80 × 25 monochrome
7,6	Number of disk drives, where 00 = 1, 01 = 2, 10 = 3, 11 = 4
8	Not used
11,10,9	Number of RS-232 ports attached
12	Game I/O attached
13	Not used
15,14	Number of parallel ports (printers) attached

Figure 17.2: *Equipment Flag Values*

The COMPAQ's Ever-Active Timekeeper

This might not be terribly thrilling, but you can look at the two timer locations in the COMPAQ's ROM BIOS data area, at 0:046C and 0:046E. 0:046C is the more active.

Exploring IO.SYS

The first few locations of many programs are JMP instructions to particular functions within. IO.SYS (also known as IBMBIO.COM) is such a program. It can be instructive to follow those jump instructions and see what a program does and how it works. Following the theme of a program is often easy when the program makes frequent use of the interrupt 21H function calls. You can tell what the program is doing by looking at the value of the AH register when it executes the INT 21 command.

Recovering Erased Files

This operation is for the adventurous. When the Erase (or Del) command has done more than you wished, you can recover the lost file(s) using DEBUG and what you know about the directory structure and file allocation tables.

It is best to have practiced this recovery operation before you need it (or have purchased commercially available software that does it). A bad time to attempt your first recovery operation is late at night when you are tired and angry that a valuable file has disappeared.

The recovery operation has a much greater chance of being successful if no more work was done on the disk after the file was accidentally erased. If that is true, the erased directory entry is the place to begin. The first byte of the name has been changed, but the rest of the directory entry, including the address of the first cluster, is still valid. The blocks belonging to the erased files have been allocated, so you will need to reconstruct the FAT entries. It is not sufficient to fix the name in a directory entry, even if this makes the file reappear. The FAT still thinks the space is free, and the block(s) will soon be allocated to another file. CHKDSK will reveal this incomplete recovery condition.

Once more files have been added to a disk with an erased file, your chances of recovering that file are quite small. Again, CHKDSK will help in determining the recoverability of a file.

If you know about clusters and disks, directory entries and FATs, and you know how to use DEBUG, you can recover just-erased files.

DEBUG COMMAND SUMMARY

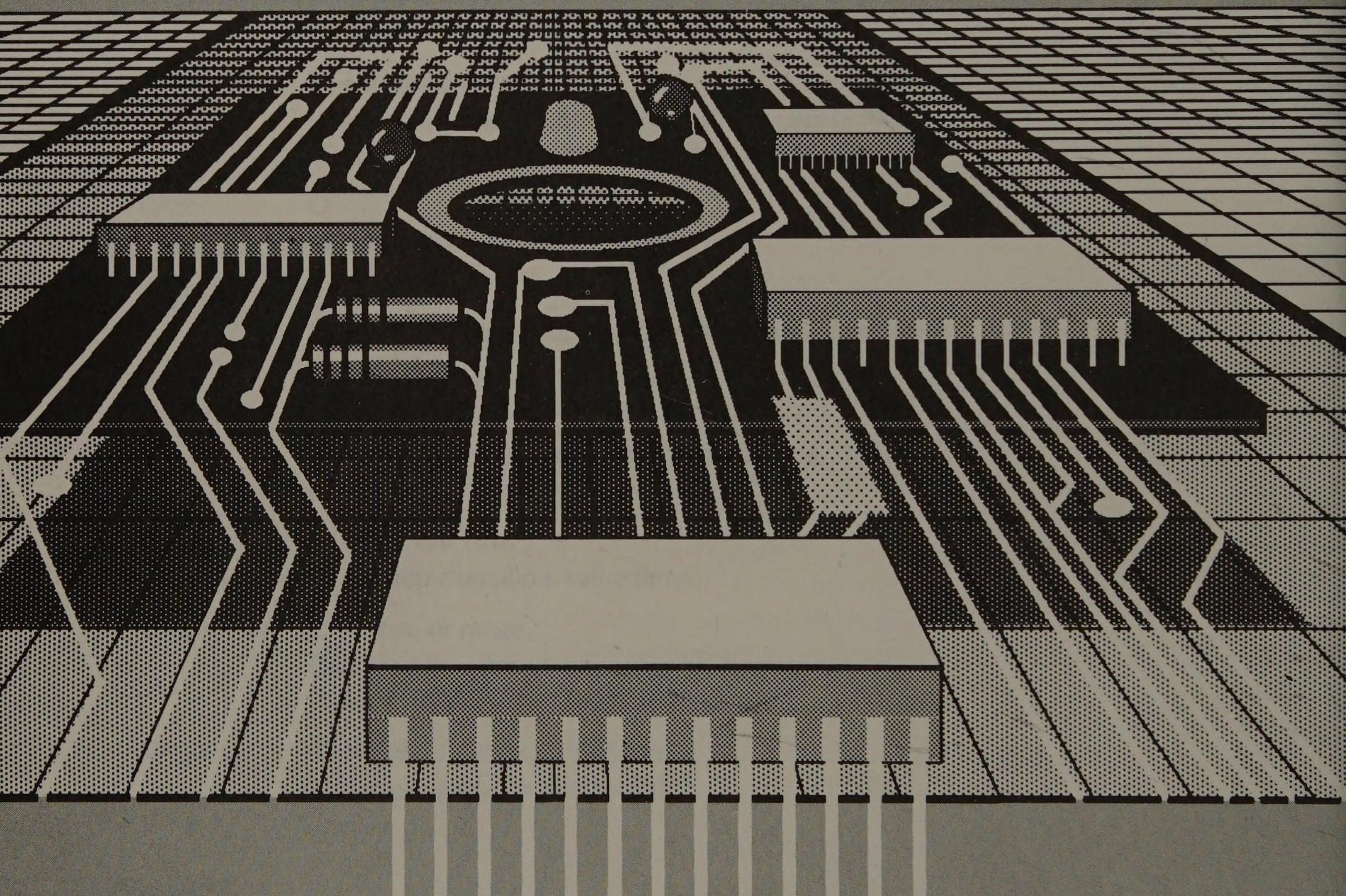
DEBUG commands are single characters, followed by their arguments, typically the range and/or address on which they operate. Commands are presented in alphabetical order.

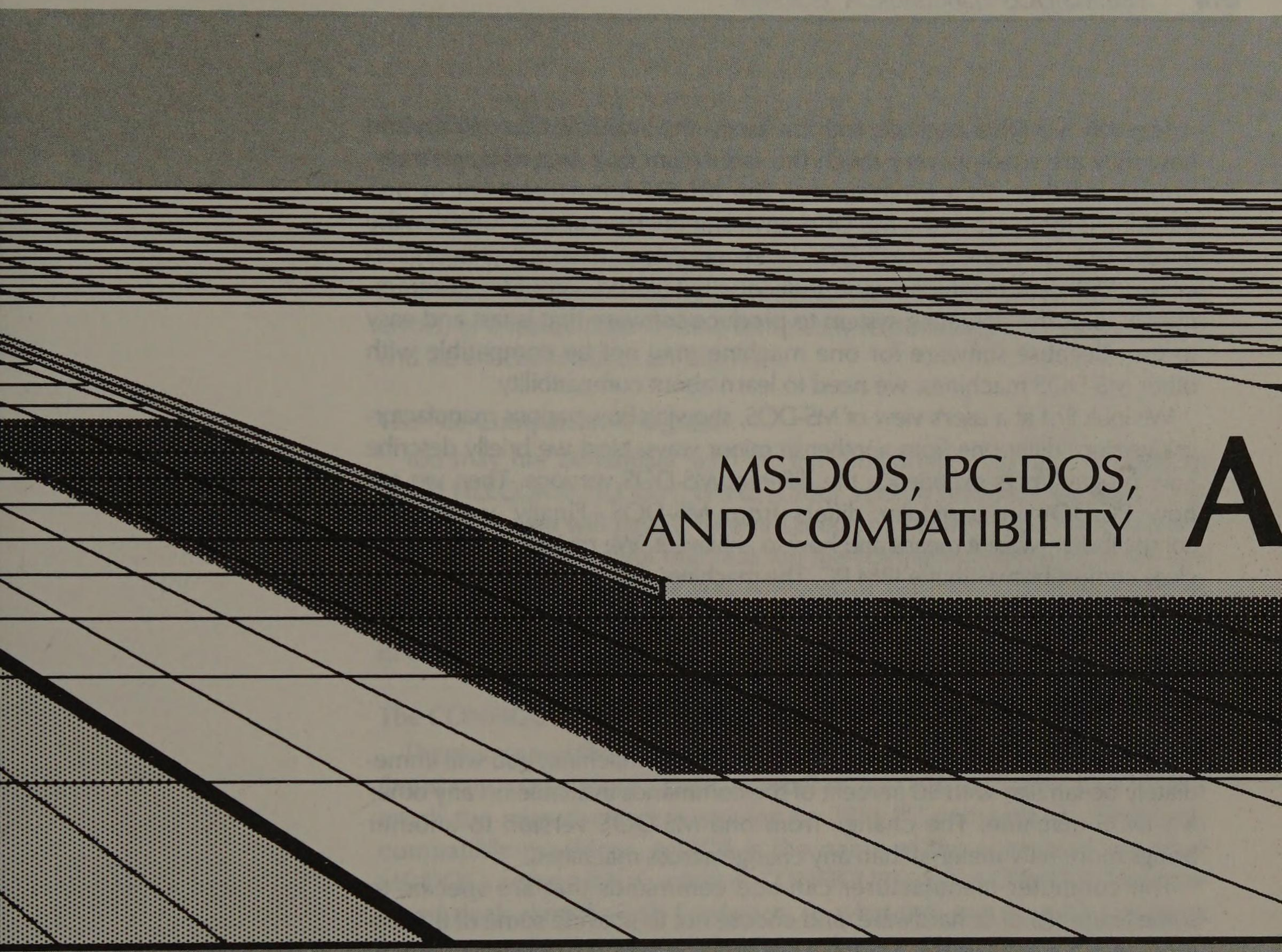
Command	Format and Notes
Assemble	A *address* Assembles your (small) program at the address you specify
Compare	C *range address* Compares a range of memory to that at the address specified
Dump	D *address* or *range* Dumps memory in both hex and ASCII formats
Enter	E *address (list)* Enters data from *list* into memory, or allows byte-by-byte display, then entry of data
Fill	F *range list* Fills memory, repeatedly if necessary, from a list of bytes
Go	G *(=address), address,* etc. Executes code, optionally starting at *=address,* permitting up to ten breakpoints
Hex	H *value* value Adds and subtracts two hex numbers
Input	I *portaddress* Reads a byte from a port
Load	L *address drive sector #sectors* Loads into *address* the number of sectors specified; if no source, uses filespec at CS:80
Move	M *range address* Moves a range of bytes to an address
Name	N *filename* Defines the filename (and drive and path) for Load and Write commands

Command	Format and Notes
Output	O *portaddress* Writes a byte to an output port
Proceed	P *(=address) value* Proceeds without stopping through execution of subroutine, interrupt, loop, or string instructions
Quit	Q Exits from the DEBUG program
Register	R *register* Displays and permits alteration of all or specified registers and flags
Search	S *range list* Searches for a *list* of characters over a range of addresses
Trace	T *(=address) value* Single-step execution, *value* times
Unassemble	U *address* or *range* Disassembles memory into assembly language instructions
Write	W *address drive sector #sectors* Writes programs or data to disk

SUMMARY

In this chapter you have learned how to use DEBUG to look into MS-DOS, to patch programs, and to create small programs. DEBUG is a powerful tool in the hands of a knowledgeable programmer; it can be quirkily destructive in the hands of a novice. DEBUG is your window into the system. Have fun.





MS-DOS, PC-DOS, AND COMPATIBILITY

A

INTRODUCTION

Microsoft's customers for the MS-DOS operating system are companies that build 8086/8088-based microcomputers. IBM, Hewlett-Packard, ITT, Digital Equipment Corporation, Compaq, Zenith Data Systems, Kaypro, NorthStar, Altos, and more than fifty other companies have acquired MS-DOS from Microsoft and offer it to their customers. Each implementation of MS-DOS is different from the others. How different, and how that difference might affect us, are the topics of this appendix.

The way MS-DOS interacts with the user—the available commands, and how they are used—is very much the same from one manufacturer's version to another. To a programmer, the MS-DOS environment in one machine is the same as the MS-DOS environment in another. However, the hardware is different, and MS-DOS is not capable of masking all the differences. Different machines need individually tailored software; programmers bypass the operating system to produce software that is fast and easy to use. Because software for one machine may not be compatible with other MS-DOS machines, we need to learn about compatibility.

We look first at a user's view of MS-DOS, showing how various manufacturer's versions differ one from another in minor ways. Next we briefly describe how programmers experience the different MS-DOS versions. Then we see how PC-DOS in particular differs from MS-DOS. Finally, we discuss compatibility—what it means and how to achieve it. We review the criteria for close compatibility with the IBM PC. The machines that meet these criteria are referred to in this book as "the IBM PC compatibles" [PCCs].

THE USER'S VIEW OF MS-DOS

Once you have learned to use MS-DOS on one machine, you will immediately be familiar with 90 percent of the commands available on any other MS-DOS machine. The change from one MS-DOS version to another brings more new material than any change across machines.

The computer manufacturer can add commands that are specific to some feature(s) of its hardware, and choose not to provide some of the features that are part of the MS-DOS package offered by Microsoft, so you won't necessarily find all the same commands in each manufacturer's version of MS-DOS.

Commands

The commonly used commands are found in every manufacturer's version of MS-DOS—COPY, TYPE, DIR, DEL, DATE, CHKDSK, TIME, and DEBUG. The directory-tree commands, VER, VERIFY, and VOL are also universal. You will find PRINT, and probably GRAPHICS too, although their operation might be slightly different. Batch files are part of the power of MS-DOS; you will find them in all versions. EDLIN, the depressingly primitive editor, is always found.

The names used for commands can vary a little from one manufacturer to another. The command for copying whole disks is usually called DISKCOPY, but you might find DCOPY or DSKCOPY instead. The command that transfers system files from one disk to another might be called

either SYS or SYSCOPY. Some familiar commands have more or less power than expected, such as FORMAT.

The FORMAT Command

In MS-DOS versions that are not almost identical to PC-DOS, the FORMAT command may have a few extra options. You might be able to display soft errors, or initialize disks without completely reformatting them (the directory and file allocation tables are cleared).

The File-Comparison Program

You may not be familiar with the file-comparison program. It might be called FILECOMP, FILCOM, or FC. If your version of MS-DOS is very close to PC-DOS, you will have a different program, called COMP. COMP is less complex and less useful than the full file-comparison utility. COMP's primary purpose is to show you whether or not two files are the same. FILECOMP was built for programmers; it gives more detail on how files differ, and allows for files of different length.

The CONFIGURE and MODE Commands

The command that allows you to configure the serial port parameters (baud rate, parity, word length, number of stop bits) is another clue to how much the manufacturer is striving for IBM PC compatibility. In closely compatible machines, MODE is the name of the command. In other MS-DOS versions, it is a variant of CONFIGURE—often CONFIG. This command deals directly with hardware, so what you can do with it is more likely to vary from one computer to another.

Additional Commands

Most of the additional commands are tools for software development, such as APPLY and MAKE. You may find RDCPM, which reads files written to disks under the CP/M operating system. Hardware-specific commands may exist. For example, the command FONT may allow you to change character sets.

The File System

MS-DOS manages files on disk. These capabilities do not differ from one manufacturer's computer to another. File names are specified in the same way. Rarely, some nonalphanumeric reserved characters may differ from DOS to DOS. The reserved filenames and extensions are the same, as is the handling of wild-card characters (? and *). However, this does not mean that disks written under MS-DOS on, say, a Kaypro 2000 can be read on the

IBM PC. The Kaypro 2000 disk drives use 3½-inch disks, so file transfer has to be done through a communications link instead of media transfer.

The Prompt

The prompt message is another, minor source of variation among different manufacturers' versions of MS-DOS. For example, the PC-DOS prompt is C> (if the default drive is C). In another MS-DOS it may be C:. The DEBUG prompt is – in PC-DOS, > in another MS-DOS. If you don't like your manufacturer's choice, you can use the PROMPT command to create your own variation.

Documentation

The documentation provided by the manufacturer varies considerably. Microsoft's documentation for MS-DOS consists of a *User's Guide,* a *Programmer's Reference Manual,* and the *Utility Software* documentation which describes utility programs provided mostly for programmers. It's fairly well written, but definitely can be improved in places. Some manufacturers pass along the best sections provided by Microsoft and skip the rest, others rewrite extensively and provide complete documentation.

THE PROGRAMMER'S VIEW OF MS-DOS

Programmers need two different sets of facilities: good software development tools, and plentiful functions that their programs can call to use the hardware.

Software Development Tools

Microsoft supplies a set of development tools that assist programmers in writing applications that run under MS-DOS—a macro assembler, a linker, a librarian, and cross-referencing and debugging packages. APPLY and MAKE can be added to the list. APPLY and MAKE are UNIX-derived commands; they are specialized members of the batch operations family of software development tools.

Not all these tools are passed along to the purchaser of MS-DOS. For example, IBM charges an extra $100 for the macro assembler, and does not supply the librarian and cross-referencing tools. The presence of commands such as MAKE and APPLY and the inclusion of the macro assembler and librarian package are good indicators of the degree to which the manufacturer expects to accommodate programmers.

Function Calls

The manufacturer leaves intact the functions that can be called by programs (the MS-DOS standard interface features). Otherwise, the manufacturer's purpose in acquiring MS-DOS would be defeated.

In all manufacturers' versions, the MS-DOS interrupts and function calls are the same—not just apparently the same, but truly so. The arguments are passed in the same registers, the error codes are the same, and the results (file manipulation, character I/O, etc.) are the same.

File-Access Differences

A prime function of MS-DOS is to provide file and disk-handling functions for programs. These functions are the same for all manufacturers' versions of MS-DOS. (Sometimes additional access functions exist, but they are additions, not replacements.) File control blocks are used in the same way, and their contents are identical. Disk directory entries are the same, and the layout of the disk is the same—boot record, two copies of the FAT, followed by the directory and system files.

THE DIFFERENCE BETWEEN MS-DOS AND PC-DOS

IBM's PC-DOS is just another variant of MS-DOS. Other manufacturers have also been known to call their versions of MS-DOS by different names. For example, one of the Zenith line of 8086/8088 computers uses a variant of MS-DOS called Z-DOS. Here we summarize the unique features of MS-DOS when it appears under the name PC-DOS.

In PC-DOS, the disk-copy and disk-compare commands are DISKCOPY and DISKCOMP; they are often abbreviated to DSKCOPY and DSKCOMP. The command that copies the system files to another disk is SYSCOPY, not just SYS. You can see already that the differences are slight.

In general, the commands have been made easier to use by removing subfunctions that are more appropriate for programmers than users of applications software such as spreadsheets, word-processing, and database programs. The file-comparison utility found in many versions of MS-DOS—FILCOMP—has been stripped down to a program called COMP in PC-DOS, which tells you whether two files are the same, but not how they differ in any detail. The FORMAT command has fewer options than in most versions of MS-DOS. The MODE command has options that are specific to the display types available on the IBM PC, as well as including the normal serial-port configuration commands.

IBM chose to release with PC-DOS the linker and the debugger; it sells the macro assembler separately, and does not supply the librarian. In

PC-DOS the two system files are called IBMBIO.COM and IBMDOS.COM, instead of the usual IO.SYS and MSDOS.SYS. The differences in BASIC implementations are discussed in the section on IBM PC compatibility.

To a programmer, PC-DOS is the same as MS-DOS. However, programs will often bypass MS-DOS, so programs written for the IBM PC won't necessarily run on other MS-DOS computers.

IBM PC-COMPATIBLE COMPUTERS

PC-compatibility has become an important topic, with many computer manufacturers claiming it, and much confusion as to what compatibility really means. Here we examine why compatibility with the IBM Personal Computer can be important, in what forms it is claimed, and what to consider when checking for compatibility. We see that the most important form of compatibility is software-compatibility, where software written for the IBM PC will run without flaw in another MS-DOS computer.

Compatibility—Why It Matters

In the world of 8086/8088-based computers, most software is written first for the IBM Personal Computer. If you want a general-purpose personal computer, you want one that will run the software written for the IBM PC. More than 90 percent of IBM PC users use MS-DOS. Excellent packages have been written to run under other operating systems, and on other computers, but the greatest variety of software is found running under MS-DOS on the IBM PC.

Another aspect of the current marketplace is that you can't sell computer hardware without software. A manufacturer who can build a computer that runs software written for the IBM PC will avoid enormous software-development expense. However, the IBM PC, although sturdy, reliable, and popular, doesn't have all the latest technological advances. Newer computers can be better than the PC; the manufacturers have a tough time balancing probable sales against the desire to introduce new technology.

Compatibility Claims

PC compatibility is widely claimed. It is also made hazy by the yearnings of people who sell computers based on the Intel 8086 family. The claim can be based on as little as the fact that the computer, when it is not running its regular operating system, can run MS-DOS, or, at the other extreme, the claim can be based on the fact that the new computer can run any and all software written for the IBM PC.

"MS-DOS compatible" is often used as a name for the first type of compatibility. (The potential for confusion between MS-DOS compatibility and IBM PC compatibility is not entirely to the disadvantage of people selling an "MS-DOS compatible" computer.) However, the second type of compatibility is what counts; such a computer is, ultimately, a software-compatible machine.

Compatibility that Counts

If your computer dealer can show you a list of several hundred programs written for the IBM PC that, without any patching or other adaptation, run on an MS-DOS computer, and that list includes Microsoft's Flight Simulator and Lotus 1-2-3, then that computer is compatible with the IBM PC.

Even though we use the term *software-compatible* for such computers, a considerable degree of hardware compatibility is implied. The copy-protection scheme for some versions of Lotus 1-2-3 (among other applications) depends on certain operational features of the PC's disk drive. If those hardware features are not duplicated in other computers, those other computers won't even be able to load 1-2-3, let alone run it.

Data Compatibility

Other levels of compatibility are less valuable, but cannot be dismissed. You may want to tranfer files of information from one manufacturer's computer to another. If the computers are software-compatible as defined above, there will be no difficulty in making the transfer. But even if they are not, at least you will want them to be data-compatible. Fully data-compatible computers can read disks written by other computers, and write disks that can be read by other computers.

Sometimes commercial software vendors will come to the rescue when you want to transfer files between machines that are not data-compatible. Xenocopy, for the IBM PC, reads files from, and write files to, disks of different format. Sometimes the computer manufacturer will supply software that allows you to read files prepared on other machines. Xenocopy, by the way, is so specific to the IBM hardware (it addresses specific locations in ROM BIOS), that it is one of the very few pieces of software that will not run on any PC compatibles.

Hardware Compatibility

Another category of compatibility is associated with hardware features. If pieces of the hardware are interchangeable with the IBM PC, then

programs written for the PC that use the interchangeable harware will most likely run well.

ROM BIOS

The ROM BIOS chip in the PC is unique to the PC. Other manufacturers can use the same interrupt locations as IBM to point to software functions that accept the same parameters and provide the same software services, but they cannot exactly duplicate the BIOS. Few programs make use of the routines found in the BIOS other than by way of the standard interface through an interrupt location, but when they do, the chances of those programs running perfectly on any PC-compatible is close to zero. IBM sues manufacturers who copy their ROM BIOS.

Other companies make alternative ROM BIOS chips. Their command interface is sometimes perfectly compatible with IBM's ROM BIOS, but not always. The Phoenix ROM BIOS is good.

ROM BASIC

The IBM PC has a subset of BASIC installed in a ROM chip. The disk-based BASIC (or BASICA) makes extensive use of the ROM BASIC capabilities. Programs written in BASIC on the PC will not run on computers that don't have the same set of capabilities. However, GWBASIC (also supplied by Microsoft, who wrote BASIC for the PC) is supposed to run programs written in BASIC for the PC. Some commercial software manufacturers supply AUTOEXEC.BAT files that start operation of their programs. That batch file would contain the line

```
BASICA HIGAME
```

If the batch file is changed to read

```
GWBASIC HIGAME
```

software compatibility may be regained.

Expansion Boards

Computers that are most hardware-compatible accept memory expansion boards, device-controller boards, and other add-on boards made for the IBM PC. As usual, statements that one computer will accept boards made for another computer may not mean that the two computers are hardware-compatible; the boards might fit in the slots but they may not always do what is required of them. A full discussion of hardware compatibility is outside the scope of this book.

However, if you do find an MS-DOS computer that works just fine with IBM PC boards, and vice-versa, the chances of the MS-DOS computer being software-compatible are high.

Keyboard Compatibility

The IBM PC keyboard has odd key placement, and a selection of keys not always found on other computers (e.g., Alt, PgUp, PgDn, Numlock). To run IBM software, the keyboard layout does not have to be the same (we can adapt to a difference in key positions) but access to all the keys has to be possible!

Display-Handling Compatibility

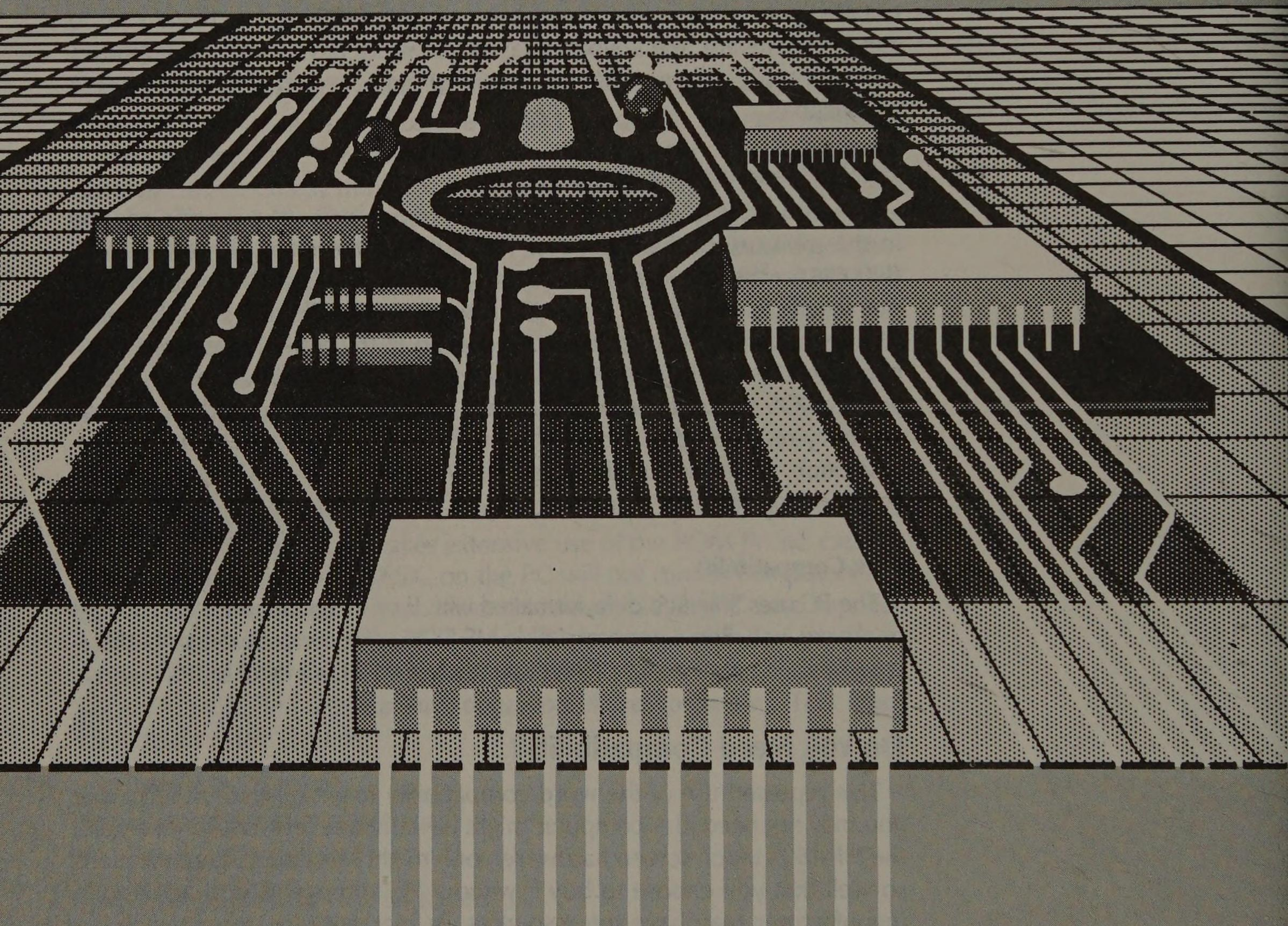
The IBM PC display gets its characters from a buffer at a specific address in the memory of the computer (B0000H for the monochrome display, B8000H for the color/graphics display). The adaptor boards are built to output specific characters of a specific resolution (see Chapter 15). The set of 256 characters used by the PC is made up of the first 128 standard ANSI characters, and a selection of IBM-chosen characters for the other 128. If an MS-DOS computer uses a different memory address, or does not use memory-mapped displays, if it forms character sets at a different dot-resolution, or sends out characters different from the IBM PC, it won't be software-compatible.

Disk Compatibility

The PC uses 5¼-inch disks formatted with 9 or 15 sectors per track, and 48 tracks per inch. Some noncompatible MS-DOS computers use disks of a different physical size, or with a different number of tracks per inch, or both.

Do You Need Compatibility?

This repeated use of the word compatibility might obscure the fact that you may not need it! A computer might have unique features (for example, the HP-150's touch-screen hardware), or it might serve specific applications so well that you choose to buy it without PC compatibility. In addition, a manufacturer may have the resources, or market-share, to convince the creators of popular software packages to tailor them for its machine. Remember that your choice of machines may not be strictly limited to whether or not they are software-compatible with the PC.



THE DIFFERENT MS-DOS VERSION NUMBERS

B

This appendix points out the differences between major versions of MS-DOS and some of the important differences between the minor within major versions. Major versions are denoted by the first digit of the version number. Minor versions are denoted by the digits following the period. The major versions are summarized first, then the minor version differences are discussed. The detail for version 2 is more extensive than for the other versions in honor of the valiant PC *jr.*

MS-DOS version 1 was written to improve significantly upon CP/M, and succeeded, but its roots were quite clear. Version 2 brought a tree-structured filing system, I/O redirection, and the first major moves toward Unix, a powerful operating system long favored by professional programmers because of its flexibility and extensibility. The movement toward Unix, perhaps toward multitasking and multiuser systems, stalled after version 2. It stalled possibly because Microsoft's and IBM's interests were not served by offering too much too soon, and possibly because implementing a Unix-like system for commercial applications on an 8088 is genuinely risky. The Unix user interface is not friendly, and in the 8088 no hardware features are available that help prevent a system-wide crash if one program runs amuck. So version 3 provides for network operations, which fairly neatly meets the customers' and computer manufacturers' needs.

MAJOR VERSION DIFFERENCES

This section provides summaries of the differences between each major version of MS-DOS and the next.

Beginning with Version 1

Version 1 is distinguished by a file access system based on file control blocks (FCBs). Because a large amount of software was written using FCBs, subsequent versions have maintained the ability to run programs that use FCB-based file access, but that support is, appropriately, fading.

In version 1 no support was provided for hard disks, just floppy disks, and even then the diskette would hold a maximum of only 112 files. That did not stop peripheral equipment manufacturers from building hard disks for the PC, but integrating them into MS-DOS software was awkward. Ramdisk support was also hard to implement. Improving the performance of version 1 systems was difficult.

Improvements with Version 2

Version 2 brought

- the official ability to use hard disks,
- a tree-structured file system,
- data access via an address within the file,
- ease of incorporation of unique device drivers,
- a limited form of I/O redirection,
- disk-based, rather than memory-based, pipes,

- the ability to configure the internals of MS-DOS,
- extended cursor control and keyboard key reassignment via the ANSI.SYS device driver,
- print spooling,
- file recovery, and
- an expanded set of batch file commands.

With version 2 the user could tailor his or her system to better support specific needs, and this version provided the programmer with enough tools to effectively manage the PC and its peripherals.

For the programmer, version 2 also brought the ability to have more than one program in memory at the same time. It also brought a limited implementation of centralized error reporting and some complement functions that were omitted from version 1 (for example, the DTA could be read as well as set). The other significant tools expansion was the addition of the Assemble command to DEBUG.

Thoroughness with Version 3

Each version has improved slightly on the last, often carrying through to completion the ideas introduced in the previous version. For example, version 3 allows you to configure MS-DOS to exist in countries that use letters, currency symbols, country-specific keyboards, and number punctuation styles other than those found in the U.S.A. In version 2 conventions for other countries could be specified, but no foreign keyboard support was available.

Version 3 brings the ability for MS-DOS computers to operate within networks of other MS-DOS computers. High-capacity diskettes are supported. The background PRINT function can be tailored to the characteristics of your specific printer and to share execution time with other resident programs more gracefully. Better commands (JOIN and SUBST) are provided for running older software that only understands diskette drives. Network path recognition of the form

```
\\serverc\myvolume\tools
```

is provided, as well as the ability to address more than six drive volumes identified by letter. Version 3 also offers the GRAFTABL command that loads the above-127 ASCII characters for color-graphics operation.

For the programmer version 3 brings much better centralized error reporting and interpretation, including recommended courses of action. This improvement was triggered by the different approach to error conditions that networking imposes on MS-DOS. File-sharing and file-locking

controls are provided, as well as tools that help your program link to, and make use of, the facilities on the net. Version 3 includes the ability to create temporary files and a new file creation function call that won't kill an existing file. DEBUG has been improved again by the addition of the Proceed command that permits quicker examination and checkout of software.

MINOR VERSION CHANGES

This section provides, for each of the three major versions, summaries of the differences between the each version number and the next.

Versions 1.0 and 1.05 (PC-DOS)

The early versions of MS-DOS were first presented to the world as PC-DOS. The evidence for versions less that 1.1 is provided by PC-DOS. Version 1.0 was a little buggy, supported only 160K and single-sided disk drives, and was poor at handling serial I/O. Version 1.05 eliminated a few of the bugs in 1.0, and about 20 fairly obvious bugs in version 1.0's BASIC and BASICA were fixed. Minor versions after that, 1.25 and up (1.1 in PC-DOS), were much more satisfactory. Version 1.05 was supplied free of charge.

Limitations of Versions 1.0 and 1.05 (PC-DOS)

Versions 1.0 and 1.05 supported only single-sided disk drives, which is now hard to believe. The TIME and DATE commands were external. Unless the TIME command was included in the AUTOEXEC.BAT file, the TIME question was not asked each time the system was booted. Also a file's directory entry in PC-DOS version 1.0 was not time-stamped, only date-stamped.

Support of serial communications was very poor. Redirecting output to a serial printer was highly error prone. Version 1.0 of the IBM Asynchronous Communications Support package was similarly ineffective.

Versions 1.1 and Up

These are the most significant additions provided by version 1.25 (1.1 in PC-DOS):

- Double-sided disk support
- Time stamping of files
- Vast improvement in serial communications support
- Faster disk accessing

- Additional capabilities within the COPY command, including file concatenation, the /B option (binary file manipulation), and the /V option (copy verification)
- The EXE2BIN command

These are the less significant, but nonetheless useful, additions and changes:

- CHKDSK reports disk space and space used by hidden files, user files, and bad sectors.
- The DIR command accepts /W and /P options.
- DEBUG and LINK have been slightly improved.
- LINK cannot successfully use commas as file separators.
- The size of resident DOS has been increased by about 250 bytes.
- The IBM manual includes a quick-reference card.

The faster disk accessing is obtained by a decrease in the time the disk routines take to wait for the head to settle after a track-to-track move. The structure of the FAT, the file control blocks, and the function calls are the same in all versions 1. A directory entry is the same, except that the two bytes used to record the time in version 1.25 (1.1 in PC-DOS) are marked reserved in version 1.0.

Versions 2.0 and 2.1 [IBM PC]

Version 2.0 was, as is typical for an x.0 release, a little buggy. Version 2.1 cleared up the bugs. Also, in PC-DOS, version 2.1 is adapted to the PC*jr*. For this reason, the information provided about the version 2 differences is more extensive than that provided for other versions. The evidence for these version differences is taken from the IBM releases of PC-DOS. Later issues of MS-DOS for other manufacturers may have the PC-DOS 2.0 bugs fixed, yet still be called 2.0.

PC-DOS 2.1 is functionally the same operating system as 2.0. The user experiences the same set of commands, and an applications program sees the same operating system in both 2.0 and 2.1. PC-DOS version 2.0 had been available for about nine months before version 2.1 was introduced to coincide with the arrival of the IBM PC*jr*. IBM says you need version 2.1 to run the PC*jr*. Version 2.1 runs on the PC*jr*, the PC, the PC Portable, and the PC XT.

We also describe the few functional differences between version 2.0 and 2.1, the characteristics of programs that run on both the PC and the PC*jr*, and finally, the dialects of BASIC supplied with version 2.1 and the PC*jr*. The software and documentation provided in version 2.1 are described separately.

Version 2.x Software

IBM has chosen to provide maximum software compatibility between the PC, the PC Portable, the XT, and the PC*jr*. If a program doesn't use features unique to any one of the four types of PC, fits in the available memory, and is well-behaved, it will run on any of the four types of PC. A well-behaved program uses only MS-DOS and ROM BIOS function calls for its I/O. In consequence, a variety of applications programs that run on an IBM PC under version 2.0 also run under version 2.1, on both the PC and the PC*jr*.

Versions 2.0 and 2.1 are each delivered on two single-sided diskettes called DOS and Supplemental DOS. Directories of each version show the same files except that the 2.1 Supplemental diskette has one extra file, MUSICA.BAS. Two-thirds of the programs supplied on the version 2.1 diskettes are identical to their counterparts on the 2.0 diskettes; the rest are only slightly modified.

Version 2.1 contains about 2560 more bytes of program code than version 2.0. The increase is found mostly in the programs that have to be aware of the operation of the PC*jr*'s disk controller (FORMAT, DISKCOPY, DISKCOMP, and SYS). COMMAND.COM in version 2.1 will occupy 128 more bytes of memory than in version 2.0. That is a small increase, and will tip very few existing programs into oblivion.

The Supplemental diskette contains the SAMPLES.BAS programs that play music, draw pie charts, and let you run race cars into donkeys, although you score points for not doing that. Users of PC-DOS version 1 may be familiar with these BASIC demonstration programs. PC-DOS 2.1 contains MUSICA.BAS, a program that takes advantage of the additional sound output features of the PC*jr*. Three programs that most people rarely use are also found on the Supplemental diskette: EXE2BIN.EXE (which converts some .EXE files to .COM files), LINK.EXE (which links together files generated by language compilers), and DEBUG.COM. DEBUG may be fun to play with, and is useful for patching programs like WordStar into civilized performance.

Documentation

PC-DOS version 2.1's documentation is better organized than its predecessor. It presents at the beginning the subjects of I/O redirection, pipes, and filters—major features that distinguish version 2 from version 1. It then plunges ahead with the command descriptions. This organization encourages users to make use of the distinctive features of version 2, and gives anyone past the beginner stage quick access to useful information. The instructions on how to configure your system for optimum performance have been brought forward. Users might now read that section before information overload occurs.

Version 2.0's documentation is provided all in one manual. Version 2.1's documentation is presented in three manuals: the *User's Guide,* the DOS manual, and the *DOS Technical Reference* manual.

The *User's Guide* is a one-hundred page introductory booklet. It features color cartoons of a large-beaked chicken experiencing the troubles and delights of learning about keyboards, floppy disks, and backup. It is a little brighter than the usual dry but solid presentation that IBM favors.

The DOS manual is the major item of documentation. The eight chapters and six appendices contain all the information needed by most DOS users.

The version 2 *DOS Technical Reference* manual is available for an additional fee (this Technical Reference is not to be confused with the hardware Technical Reference manuals applicable to each type of PC). The *DOS Technical Reference* contains information needed by programmers who venture beyond BASIC. The version 2.1 manual is a reorganization of the appendices, Chapter 13 (Extended Screen Control and Keyboard Key Reassignment), and Chapter 14 (Installable Device Drivers) of the version 2.0 manual.

Although the major difference between version 2.0 and 2.1 manuals is one of organization rather than content, there is new information to be found in version 2.1. The error message descriptions (Appendix A of the DOS 2.1 manual) have been expanded to include "action" notes. The notes suggest actions you might take to remedy the situation producing the error. A short but useful chapter (Chapter 10) has been added to the technical material. It explains how DOS memory management operates and shows how to use the three memory management function calls (48H, 49H, and 4AH).

A few errata have been carried over from the version 2.0 manual. The function numbers on the first two lines of the function call summary on page 5-12 of the *DOS Technical Reference* are expressed in decimal (12), instead of hexadecimal (0C). A similar mistake has been made in the example on page 7-3 (same manual) of how to use the EXEC function to execute commands from within an applications program. The example reads

```
DB 14,"DISKCOPY A: B:",13
```

It should read

```
DB 0E,"DISKCOPY A: B:",0D
```

The memory block definition for function 38H has been corrected.

Functional Differences between PC-DOS Versions 2.0 and 2.1

Version 2.1 includes fixes to flaws found in version 2.0. Version 2.0 does not properly keep track of hard errors on the hard disk. It repeatedly tries to use

those faulty areas of the disk; the user sees a lot of disk error messages. Version 2.1 remembers the faulty spots, and does not try to use them.

It is reported that CHKDSK in version 2.0 will on occasion die when asked to work on diskettes not formatted under version 2.0. Version 2.1 notifies you of the difference in versions instead of dying.

It is also reported that version 2.0 has trouble with half-height disk drives. Version 2.1 is apparently more tolerant of their characteristics.

The SORT filter contains more code in version 2.1, although the benefit to the user is not immediately obvious.

Programs That Run under Version 2.1 on the PC and PC*jr*

You can buy, or write, programs that run equally well on the IBM PC, the PC Portable, the PC XT, and the PC*jr*. The four things you have to watch out for are:

1. Screen access. There is no "monochrome" screen on the PC*jr*.
2. Memory requirements. A program may not occupy more than 112K.
3. Disk access. There is no DMA transfer on the PC*jr,* and there is only one disk drive.
4. Code that depends on instruction timing.

It is possible for the same program to run on all three types of PC because IBM has chosen to have the PC*jr*'s ROM BIOS service the same system interrupts as the PC and XT's ROM BIOS, and all three machines run DOS version 2.1.

There are restrictions on the size and characteristics of a program that runs under DOS 2.1 in the PC*jr*. For example, software that addresses the PC's monochrome screen or needs more than 112K won't run on the PC*jr*. Once these restrictions have been met, and except for any effect that the marginal increase in size of COMMAND.COM might have, any well-behaved program that runs under version 2.0 on the PC or XT will run equally well under version 2.1 on the PC*jr,* the PC, or the XT.

The hardware in the PC*jr* appears to a program to be the same as the PC's hardware, as long as the program doesn't delve too deep. A program is considered not well-behaved when it uses the BIOS code other than by way of the defined function calls, or accesses the hardware without going through the DOS function calls.

Screen Access

The PC*jr* uses only the color/graphics device; it cannot address the standard IBM "green" monochrome screen. In addition, the memory for the

characters shown on the screen of the PC*jr* comes out of the PC*jr*'s main memory. It is not supplied separately on a color/graphics adaptor board as it is with the PC.

The color/graphics interface on the PC*jr* is still bit-mapped and still located at address B8000H. IBM has recognized that most applications programs bypass the DOS function calls that output to the screen (the function calls are quite inadequate). Most programs access the screen directly. When a program moves a character to location B8000H in either the PC or the PC*jr,* it will show up immediately in the top-left corner of your screen. This is one example where a program is not well-behaved, but is still able to run on both the PC and the PC*jr*.

The video I/O routines within the PC*jr*'s ROM BIOS (interrupt 10H) will accept three extra graphics modes: mode 8—160 × 200 pels, sixteen colors; mode 9—320 × 200 pels, sixteen colors; and mode A—640 x 200 pels, four colors. Mode 7, monochrome screen, is not a valid mode to the PC*jr*. Programs that are to run on the PC as well as the PC*jr* cannot use modes 7 through A.

Memory Requirements

The memory available to a program running on the PC*jr* depends on the video I/O modes it uses. Also, the construction of the PC*jr* requires that the 64K memory-expansion unit be added to the PC*jr* if an 80-column display is to be produced. The PC*jr* may have either 64K or 128K of main memory.

A minimum of 16K of main memory is devoted to the screen buffer. High-resolution four-color images (mode A) use 32K of the PC*jr*'s main memory. This information will help you determine if a program will run on the PC*jr*.

Neither of the PC*jr*'s BASIC interpreters is resident in main memory. The PC's BASICs take either 10K or 14K of main memory.

Disk Access

The PC*jr*'s disk I/O is not run through direct memory access (DMA). DMA operates by stealing machine cycles to transfer data to and from memory. Other machine life, including instruction execution, continues as normal; it runs a little slower because, say, every fifth cycle is missing.

Code running without DMA on a PC*jr* has to wait for completion of disk transfer operations before doing anything else, such as attending to the keyboard or a serial port. Some software, particularly that which uses the serial port, needs to be aware of this difference. The other great difference is that, for now, the PC*jr* only supports one disk drive.

Instruction Timing

The PC*jr*'s hardware *Technical Reference* manual devotes much space to warning us about timing differences between the PC*jr* and the PC. The

manual advocates using the clock instead of instruction execution time to measure short time intervals. That's like urging people who need flame to use matches instead of rubbing two sticks together. However, there are inexperienced programmers out there. It is they who need to heed this warning.

The reliable way of writing code that will run on the entire PC family is to use the DOS version 2 and ROM BIOS function calls, and to keep in mind the amount of memory absorbed by the screen image. Calls to the DOS and ROM BIOS routines have the same effect in all three members (four, if you count the PC Portable) of the PC family. If you don't address the monochrome screen, don't use features unique to the PC*jr,* and don't need more than 112K of memory, these well-behaved programs will run on both the PC and the PC*jr.*

BASIC Language

The PC*jr* may run either Cassette BASIC or Cartridge BASIC. It is not the presence of DOS version 2.1 that controls what sort of BASIC is available on the PC*jr,* but whether or not Cartridge BASIC is plugged into either of the two cartridge slots. Cassette BASIC is a limited version of BASIC built into the PC*jr*'s ROM; it will access cassettes, not diskettes. To access diskettes and other unique features of the PC*jr,* you need Cartridge BASIC.

The BASICA.COM software supplied with both DOS version 2.0 and version 2.1 is BASIC release 2.0 for the PC, the PC Portable, and the PC XT; it is not for the PC*jr.*

BASIC Language Documentation

When you buy a PC*jr,* you receive a BASIC manual that describes both Cassette and Cartridge BASIC. That is all you will need if you are running DOS version 2.1 only on a PC*jr.* However, if you will be running DOS version 2.1 BASIC on a PC or XT, you will need to send to IBM for a documentation package called *Changes in BASIC Release 2.0.* (IBM provides a form for this purpose.) You will receive about one hundred loose-leaf sheets that need to be inserted in the BASIC release 1.1 manual (second edition, May 1982). After the insertion, you have full documentation for BASIC release 2.0.

Functional Differences between Version 3.0 and 3.1

The key difference is that version 3.1 has effective network support and version 3.0 does not. In addition, as usual, some bugs in version 3.0 have been ironed out in version 3.1. For example, WordStar 3.3 fails to recognize drive B under some circumstances in MS-DOS version 3.0, but has no trouble in version 3.1.

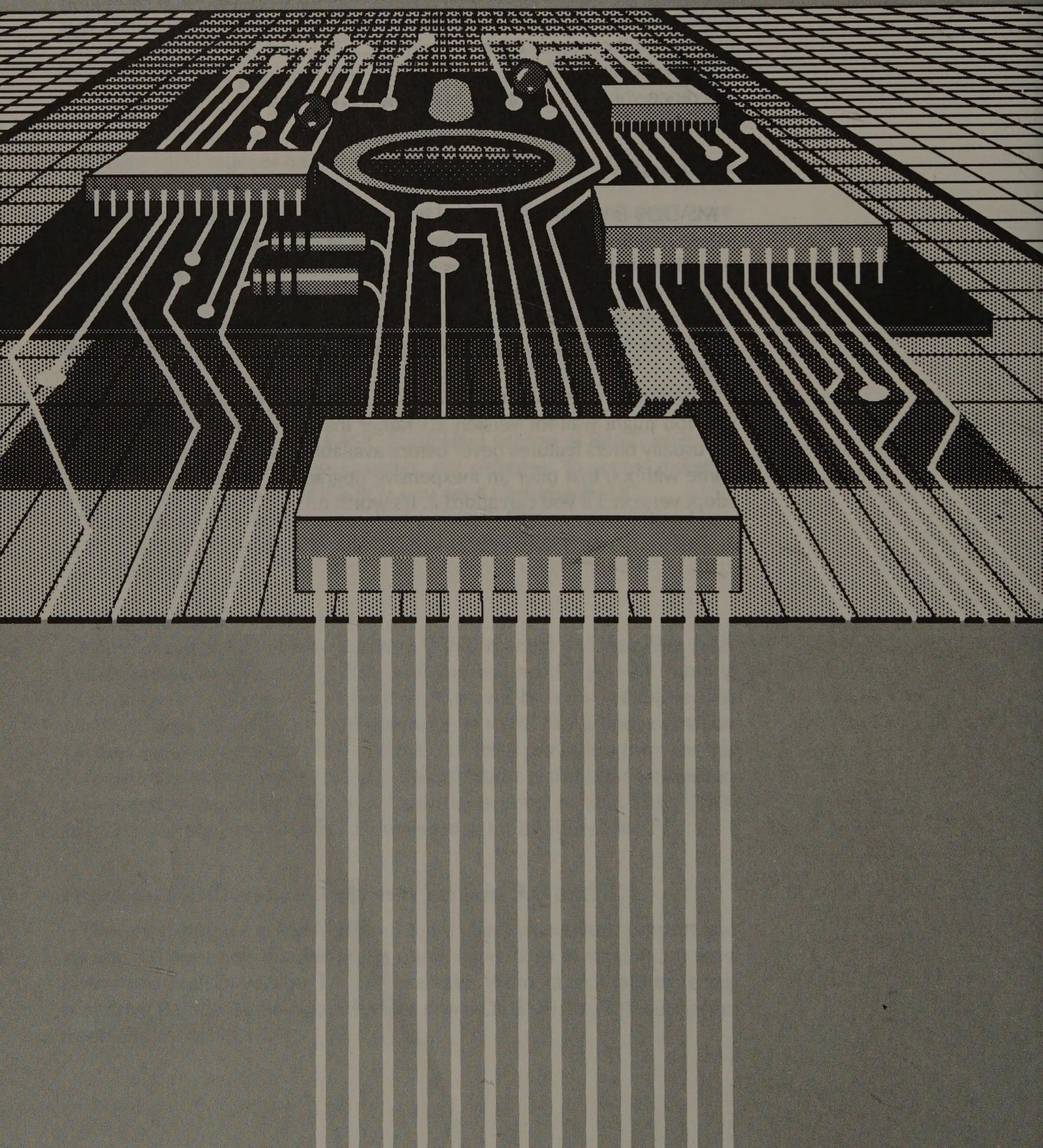
Version 3 Documentation

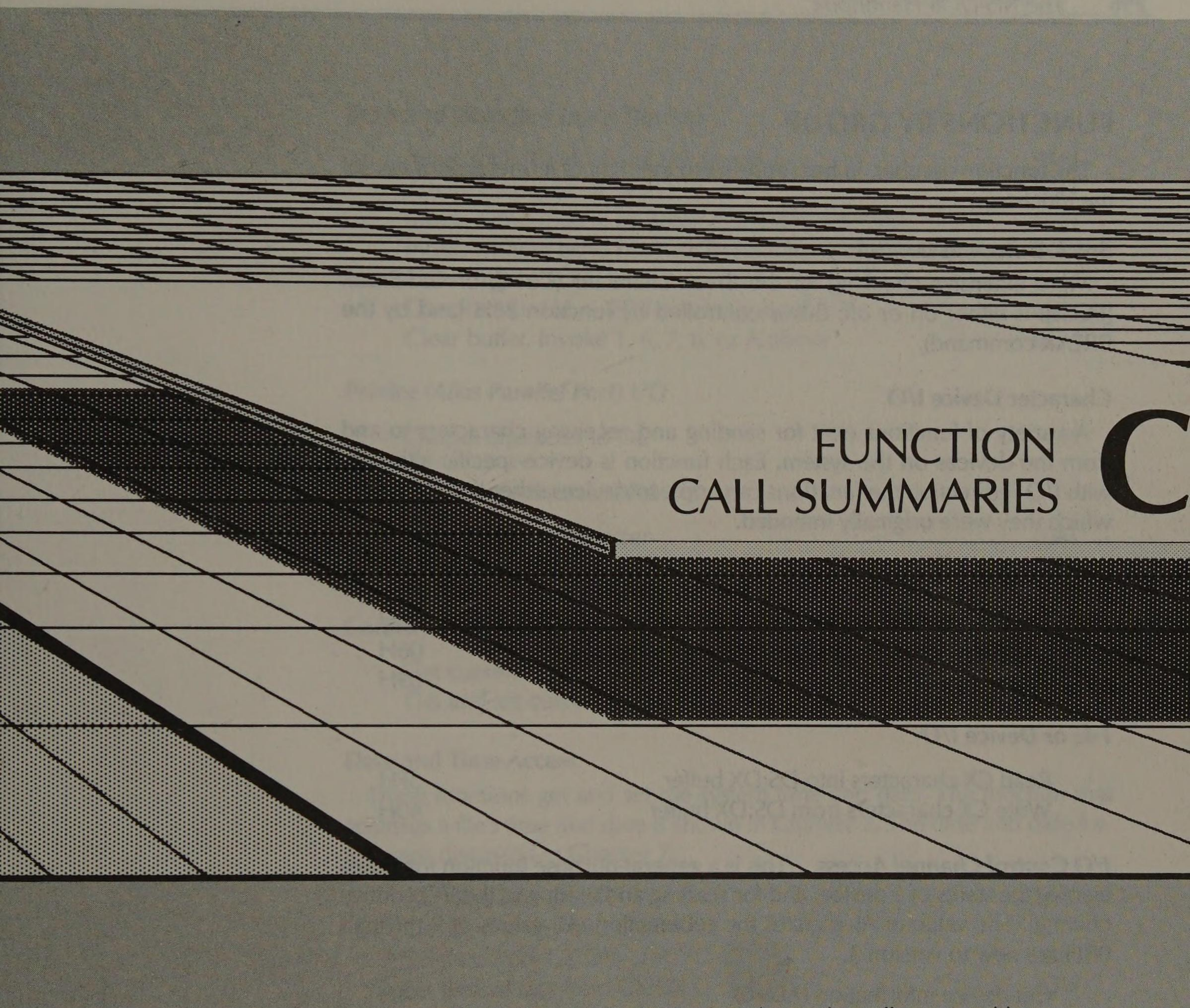
Version 3.1 user documentation is even better than that for version 2. In addition, IBM has finally produced an MS-DOS technical reference that clearly and thoroughly covers versions 2.1, 3.0, and 3.1, all in one volume. The document number is 6138536. Buy no substitutes; there is a technical reference volume for just 3.0, but it has less information.

SUMMARY

MS-DOS is continually improving, but version 3 is about as far as the MS-DOS authors might go without adding features that require significant rewriting of existing applications software. Version 3 with network support has become the new standard, although version 2.1, with its fairly full implementation of a good single-user operating system that includes tree-structured files and configurability to user-specific applications, will live long.

Experience has shown that version x.1 has fewer bugs in it than version x.0. You might wait for version x.1 rather than buy version x.0. However, x.0 usually offers features never before available; hang on to any forms that come with x.0 that offer an inexpensive upgrade to the next version. And adopt version 3 if you can afford it. It's worth it.





FUNCTION CALL SUMMARIES

C

Two function call lists are provided. The first shows the calls grouped by function, the second provides the functions in numerical order. The function requests are all made under interrupt 21H, with the function number in AX to indicate the action required (AH selects the major function, AL selects the subfunction, if necessary). Where the brief descriptions refer to keyboard and display device operations, the descriptions also apply to the standard input device and standard output devices.

FUNCTIONS BY GROUP

The function number, in hex, appears to the right of a brief description of the function.

Break-Detection Control

More extensive checking for the Break character (Ctrl-C, or Ctrl-Break [PCCs]) is either on or off; this is controlled by function 33H (and by the BREAK command).

Character Device I/O

A variety of functions exist for sending and receiving characters to and from the devices on the system. Each function is device-specific although with I/O redirection the functions can apply to devices other than those for which they were originally intended.

Display (Standard Output Device)

Output, with Break-character detection	02H
Output, without Break-character detection	06H
String output	09H

File or Device I/O

Read CX characters into DS:DX buffer	3FH
Write CX characters from DS:DX buffer	40H

I/O Control Channel Access This is a general-purpose function for investigating the status of a device, and for reading and writing to the I/O control channel. The value of AL dictates the subfunction. AL values of 9 through 0BH are new to version 3.

Get device information (AL=0)	44H
Set device information (AL=1)	44H
Read CX bytes into DS:DX buffer (AL=2)	44H
Write CX bytes from DS:DX buffer (AL=3)	44H
Read CX bytes from BL drive number (AL=4)	44H
Write CX bytes to BL drive number (AL=5)	44H
Get input status (AL=6)	44H
Get output status (AL=7)	44H
Determine if device supports removable media (AL=8)	44H
Determine if logical device is local or remote (AL=9)	44H
Determine if handle is for local or remote device (AL=0AH)	44H
Set lock conflict retry count (AL=0BH)	44H

Keyboard (Standard Input Device)

Input, with Break-character detection	01H
Attempt input (if DL = FF)	06H
Input, without echo or Break-character detection	07H
Input, without echo but with Break-character detection	08H
Get string into DS:DX + 2	0AH
Check status (AL = FF if character ready)	0BH
Clear buffer, invoke 1, 6, 7, 8, or A above	0CH

Printer (Alias Parallel Port) I/O

Output character in DL	05H

Serial (Alias Asynchronous Port)

Wait for input character	03H
Output character in DL	04H

Country-Dependent Information

Get currency symbol, date format, etc.	38H
Get and set currency symbol, etc. (new to version 3)	38H

Date and Time Access

These functions get and set the system time and date. The function that accesses a file's time and date is shown in Chapter 2. The time and date formats are discussed in Chapter 7.

Get date into CX:DX	2AH
Get time of day into CX:DX	2CH
Set date from CX:DX	2BH
Set time of day from CX:DX	2DH

Disk Access

Most disk accesses will be generated by the file-access functions. The following functions are available for more unusual circumstances.

Disk reset, flush file buffers		0DH
Read CX sectors into DS:BX	Interrupt	25H
Select default disk drive		0EH
Write CX sectors from DS:BX	Interrupt	26H

Error Code Access

Get extended error code	59H

File and Directory Access Based on File Handles

These are the preferred file access functions. File identification is by ASCIIZ string and file handle. The functions access any address within the address space of a file and manipulate files within a tree structure of files. As you can see, the majority of function calls in MS-DOS have something to do with file access.

Whole-File Access

Create a file	3CH
Create a new file	5BH
Create a unique file	5AH
Open a file	3DH
Close a file	3EH
Delete a file	41H
Rename a file	56H

Reading and Writing Data

Read from a file (or device)	3FH
Write to a file (or device)	40H
Set address within a file	42H

File Handle Manipulation

Get another file handle for the same file	45H
Point existing file handle to another file	46H

Finding Files in the Directory

Search for first entry	4EH
Search for next entry	4FH

Directory Manipulation

Make a directory	39H
Remove a directory	3AH
Change to another directory	3BH
Get current directory	47H

Utilities

Parse the file name	29H
Set the disk transfer address (DTA)	1AH
Get the disk transfer address	2FH

Get the disk parameters (free space, etc.)	36H
Get or set a file's attribute	43H
Get and set the file's date and time	57H

File and Directory Access Based on File Control Blocks

The following set of functions originated with version 1 and, for compatibility purposes, is available in all later versions. These functions require the use of correctly prepared file control blocks that specify block numbers, record numbers, and record lengths. New programs should not use these functions.

Whole File Access

Create a file	16H
Open a file	0FH
Close a file	10H
Delete a file	13H
Rename a file	17H

Sequential Record Access

Read a record	14H
Write a record	15H

Random Record Access

Read single record	21H
Read multiple records	27H
Write single record	22H
Write multiple records	28H
Set random record field in the FCB	24H

Finding Files in the Directory

Search for first entry	11H
Search for next entry	12H

File Size

Read the file size	23H
Change the file size	28H, CX = 0

Memory Management

These functions allocate and deallocate memory in units of paragraphs (16 bytes).

Allocate BX paragraphs	48H
Change size of allocated memory block	4AH
Free allocated block (segment in ES)	49H

Network Operations

These version 3 functions are relevant when file sharing is in effect.

Lock and unlock file region	5CH
Get local computer ID (AL=00)	5EH
Specify printer setup string (AL=02)	5EH
Get printer setup string (AL=03)	5EH
Device redirection (AL=03)	5FH
Get redirection information (AL=02)	5FH
Change sharing retry count (AL=0BH)	44H

Process Handling

Execute (or load) a program/process	4BH
Terminate a process (EXIT)	4CH
Get completion code of terminated process	4DH

Program Termination

Use the first two functions rather than the last three.

Terminate, return completion code		4CH
Terminate, stay resident (KEEP)		31H
Terminate, no completion code		00H
Terminate, no completion code	INT	20H
Terminate, stay resident	INT	27H

Vector Access

Two central routines for accessing the interrupt vectors are offered:

Get interrupt vector value	35H
Set interrupt vector	25H

Verify—Read after Write

It is possible to tell MS-DOS to read and check for accuracy anything that it writes to disk, using these functions (or the VERIFY command):

Select or deselect read-after-write verification	2EH
Get verify status	54H

Version Number Access

Get the version number	30H

FUNCTIONS IN NUMERICAL ORDER

Input from keyboard, with Ctrl-Break detection	01H
Output to display, with Ctrl-Break detection	02H
Wait for serial port input character	03H
Output character in DL to serial port	04H
Output character in DL to printer	05H
Output to display, without Ctrl-Break detection	06H
Attempt input from keyboard (if DL = FF)	06H
Input from keyboard, without echo or Ctrl-Break detection	07H
Input from keyboard, without echo but with Ctrl-Break detection	08H
String output to display	09H
Get string from keyboard into DS:DX + 2	0AH
Check keyboard input status (AL = FF if character ready)	0BH
Clear keyboard buffer, invoke 1, 6, 7, 8, or A above	0CH
Disk reset, flush file buffers	0DH
Select default disk drive	0EH
Open a file	0FH
Close a file	10H
Search for first directory entry	11H
Search for next directory entry	12H
Delete a file	13H
Read a record	14H
Write a record	15H
Create a file	16H
Rename a file	17H
Get current drive number	19H
Set the disk transfer address (DTA)	1AH
Allocation table address	1BH
Read single record	21H
Write single record	22H
Read the file size	23H
Set random record field in the FCB	24H
Set interrupt vector	25H
Create new program statement	26H
Read multiple records	27H
Write multiple records	28H
Change the file size	28H, CX = 0
Parse the file name	29H
Get date into CX:DX	2AH
Set date from CX:DX	2BH

Get time of day into CX:DX	2CH
Set time of day from CX:DX	2DH
Set verify status on or off	2EH
Get the disk transfer address (DTA)	2FH
Get the version number (0 if prior to 2.0)	30H
Terminate, stay resident (KEEP)	31H
Break Detection Control	33H
Get interrupt vector value	35H
Get the disk parameters (free space, etc.)	36H
Get currency symbol, date format, etc.	38H
Make a directory	39H
Remove a directory	3AH
Change to another directory	3BH
Create a file	3CH
Open a file	3DH
Close a file	3EH
Read CX characters from device or file into DS:DX buffer	3FH
Write CX characters to device or file from DS:DX buffer	40H
Delete a file	41H
Set address within a file	42H
Get or set a file's attribute	43H
I/O control, AL dictates subfunction	44H

AL	Function
0	Get device information
1	Set device information
2	Read CX bytes into DS:DX buffer
3	Write CX bytes from DS:DX buffer
4	Read CX bytes from BL drive number
5	Write CX bytes to BL drive number
6	Get input status
7	Get output status

Get another file handle for the same file	45H
Point existing file handle to another file	46H
Get current directory	47H
Allocate BX paragraphs	48H
Free allocated block (segment in ES)	49H
Change size of allocated memory block	4AH
Execute (or load) a program/process	4BH
Terminate a process (EXIT), return completion code	4CH
Get completion code of terminated process	4DH

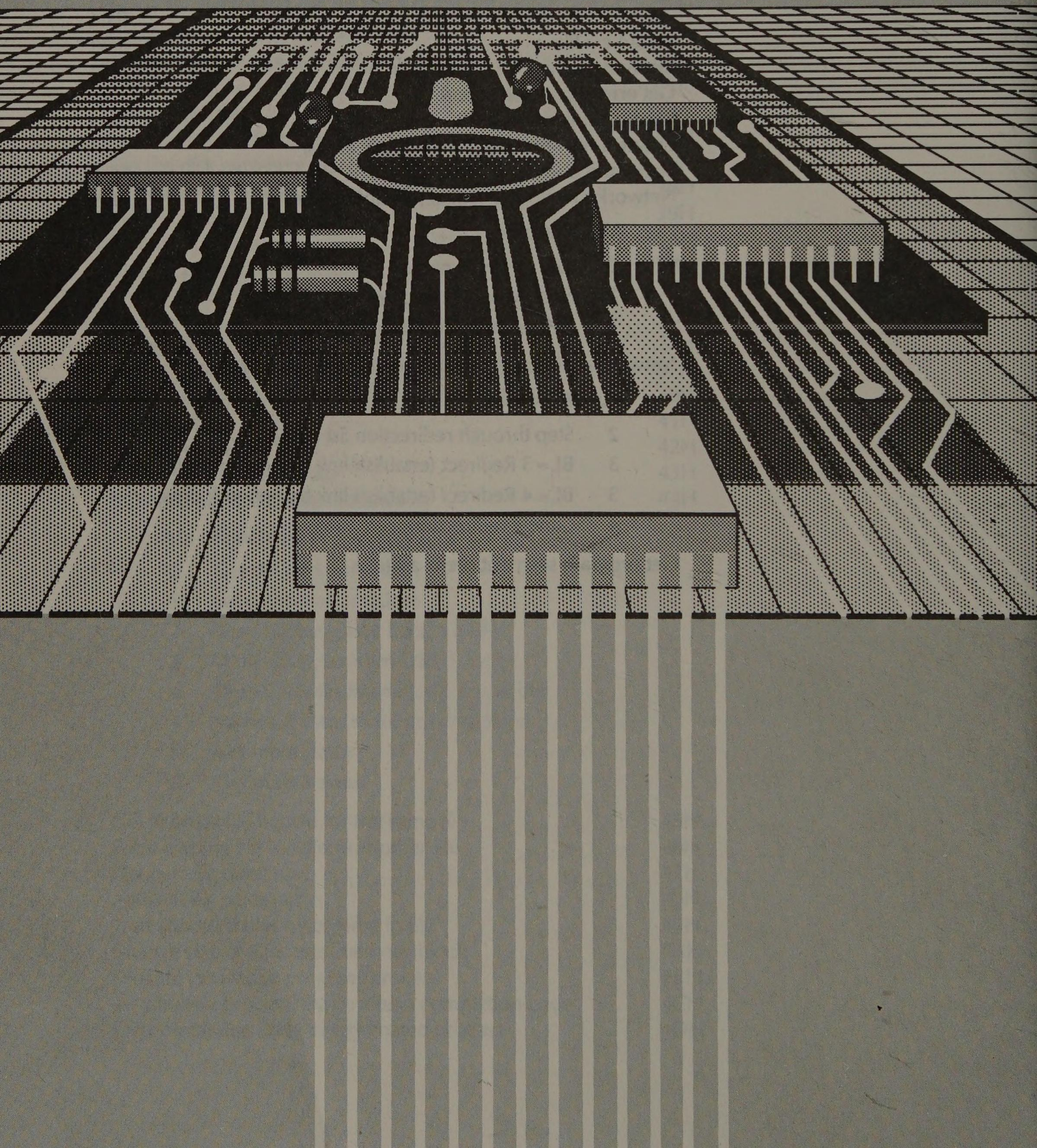
Search for first entry	4EH
Search for next entry	4FH
Get the read-after-write verify state	54H
Rename a file	56H
Get and set the file's date and time	57H
Get error condition report	59H
Create a unique file	5AH
Create a new file (fails if file already exists)	5BH
Locks and unlocks an area of a single file	5CH
Network functions, AL dictates subfunction	5EH

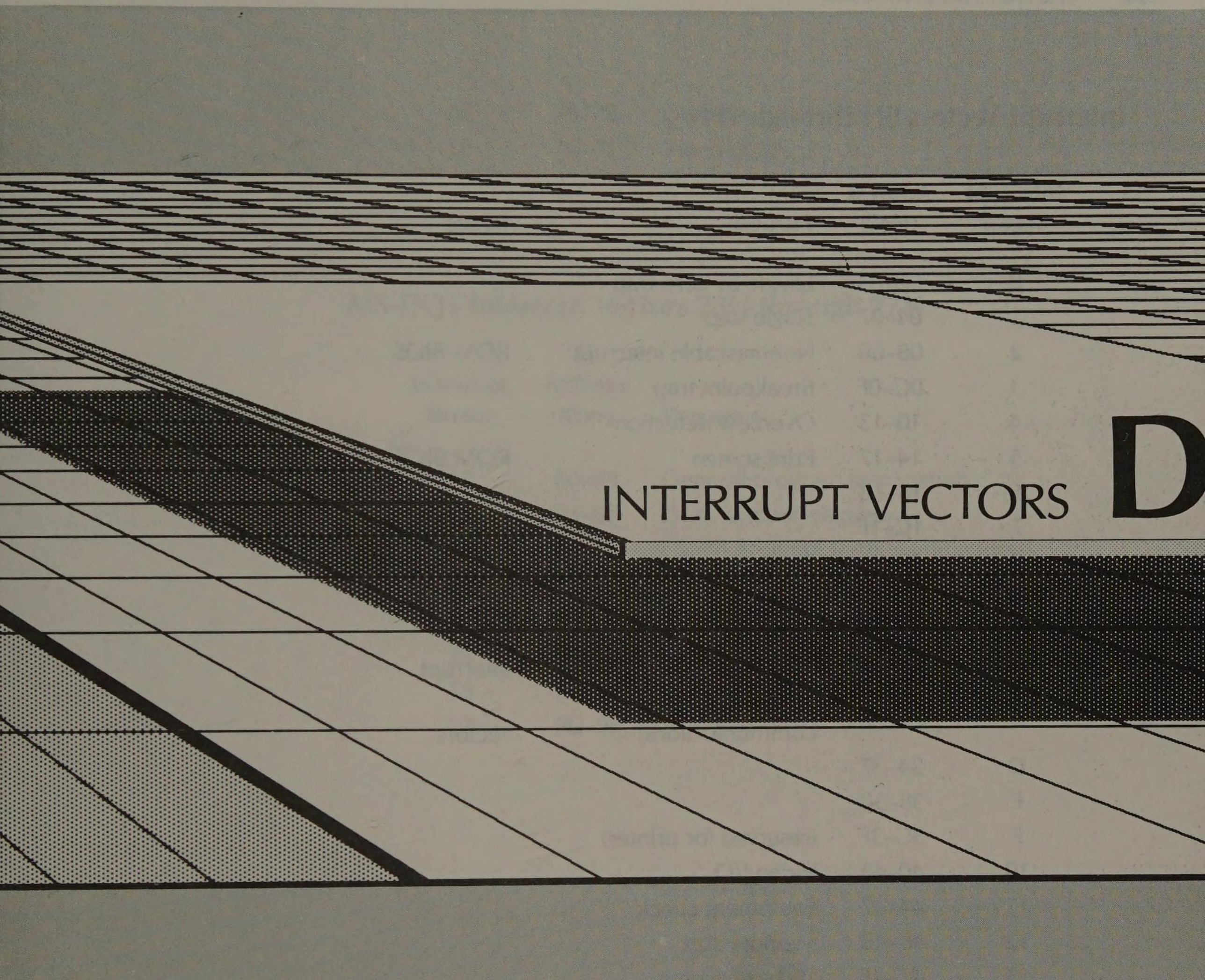
AL	Function
0	Get machine name
2	Specify printer setup string
3	Get printer setup string

Network redirection (linking), AL dictates subfunction	5FH

AL	Function
2	Step through redirection list entries
3	BL = 3 Redirect (establish link to) network printer
3	BL = 4 Redirect (establish link to) network drive
4	Cancel redirection

Get program segment prefix address	62H





INTERRUPT VECTORS D

MS-DOS claims the first 37H interrupt vectors for its own use, and initializes them when the system is booted. This appendix shows the address of each vector and the purpose MS-DOS has assigned to it. Uses for the first five vectors are defined by Intel. If your program wishes to take over the function of any of these vectors, it must replace the MS-DOS vector with one pointing to its own code. Your program must also restore the original vector when it terminates.

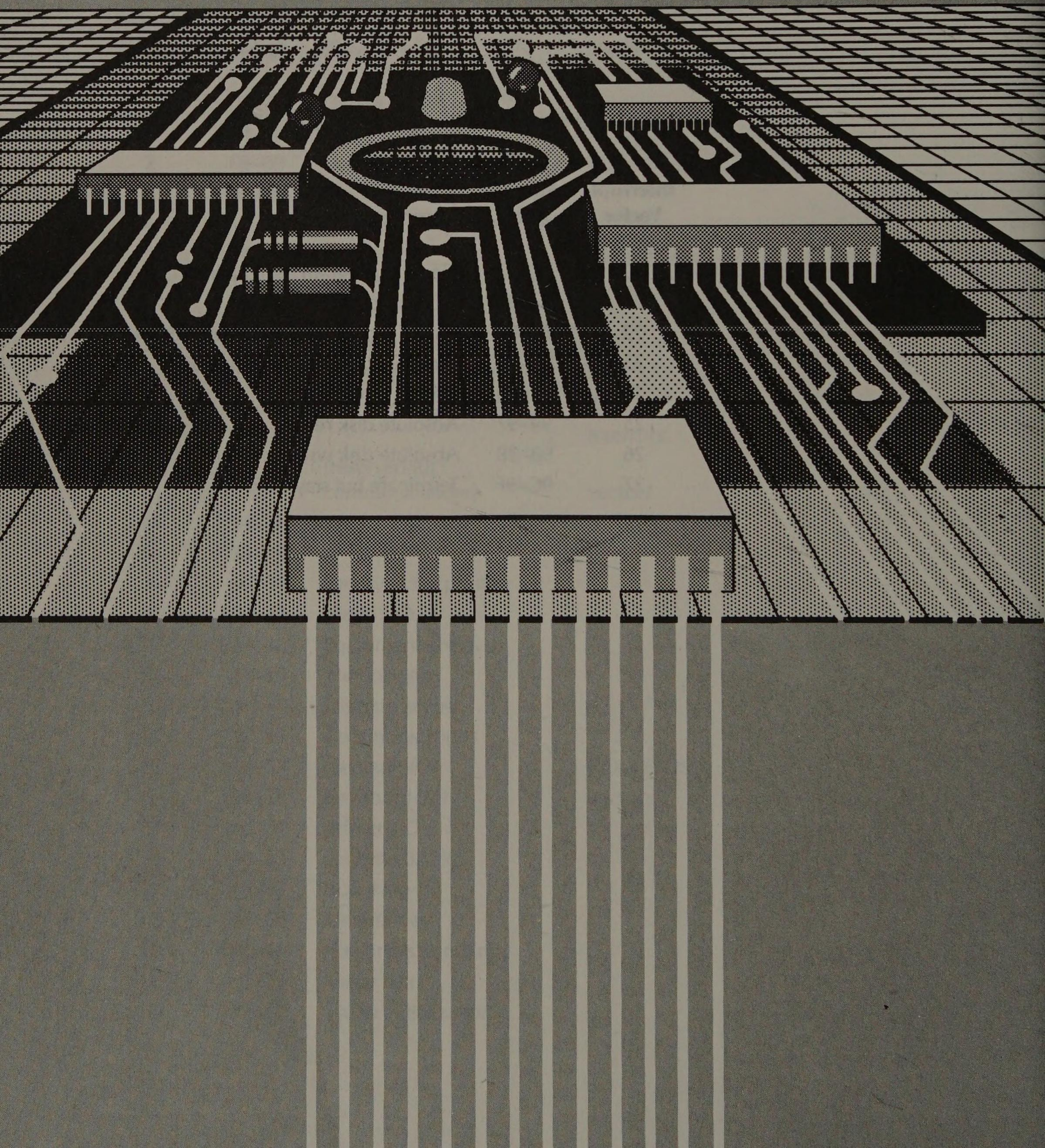
Interrupt Vectors 0H through 1FH

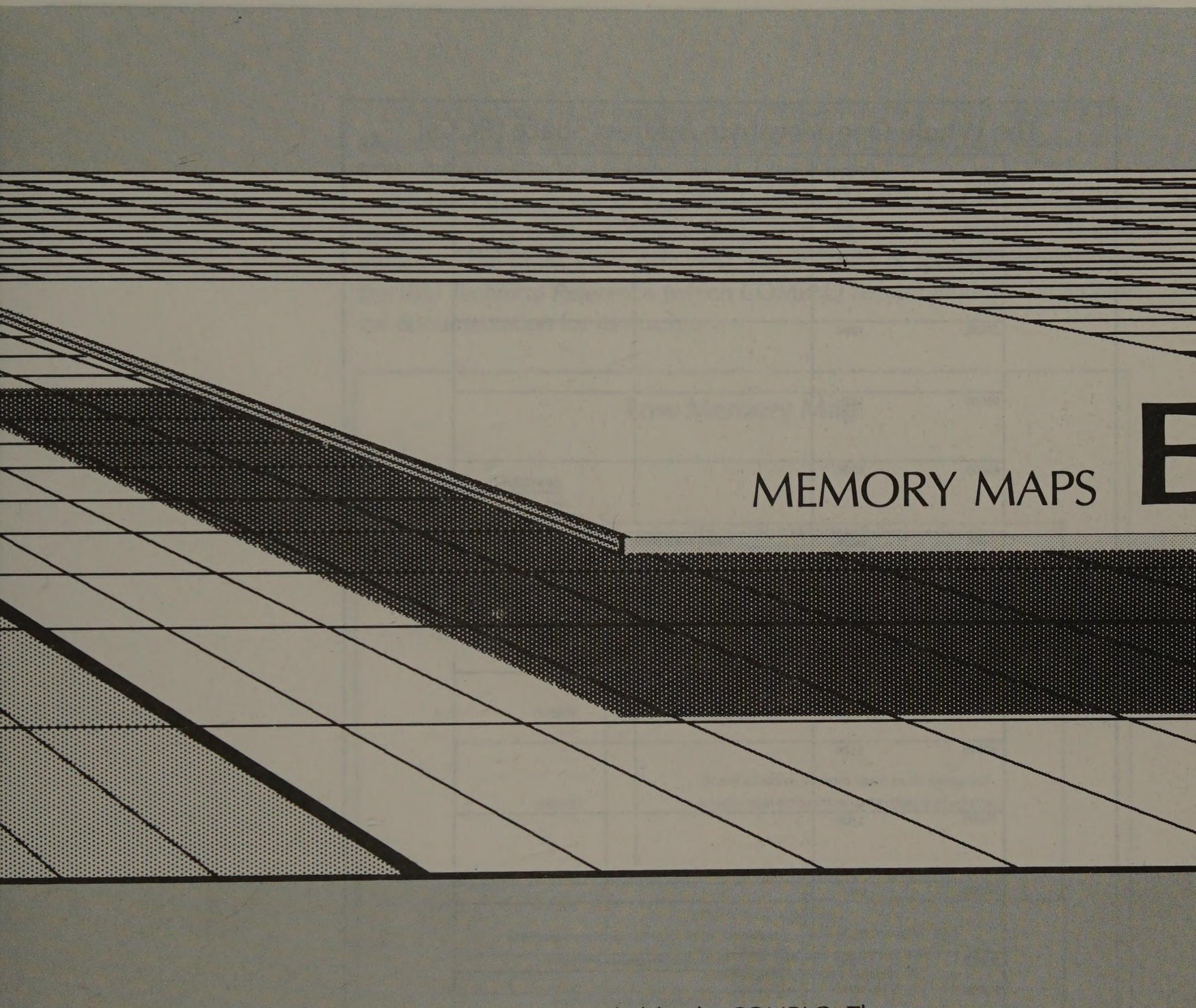
Interrupt Vector	Address (Hex)	Purpose	Notes
0	00–03	Divide by zero trap	
1	04–07	Single step	
2	08–0B	Nonmaskable interrupt	ROM BIOS
3	0C–0F	Breakpoint trap	
4	10–13	Overflow detection	
5	14–17	Print screen	ROM BIOS
6	18–1B		
7	1C–1F		
8	20–23	Timer	
9	24–27	Keyboard	
A	28–2B		8259
B	2C–2F		interrupt
C	30–33	(reserved for communications)	vectors
D	34–37		
E	38–3B		
F	3C–3F	(reserved for printer)	
10	40–43	Video I/O	
11	44–47	Equipment check	
12	48–4B	Memory size	
13	4C–4F	Diskette access	
14	50–53	Communications	
15	54–57	Cassette I/O	ROM BIOS
16	58–5B	Keyboard I/O	vectors
17	5C–5F	Printer I/O	
18	60–63	Cassette BASIC	
19	64–67	Bootstrap	
1A	68–6B	Time of day	
1B	6C–6F	Ctrl-Break handling	
1C	70–73	Timer tick	
1D	74–77	Video initialization	

1E	78–7B	Diskette parameters	
1F	7C–7F	Video graphics characters, ASCII	128–255

MS-DOS Interrupt Vectors 20H through 27H

Interrupt Vector	Address (Hex)	Purpose
20	80–83	General program termination
21	84–87	DOS function request
22	88–8B	Called program termination address
23	8C–8F	Ctrl-Break termination address
24	90–93	Critical error handler
25	94–97	Absolute disk read
26	98–9B	Absolute disk write
27	9C–9F	Terminate but stay resident





MEMORY MAPS E

Three memory maps are provided for the COMPAQ. The maps are one example of how MS-DOS uses the computer's memory.

The first map is the full one-megabyte address space map. It shows the major memory allocation at a glance, including the normal locations available for your software, the locations of ROM BASIC and BIOS, the area reserved for the ROMs of your own creation, and the memory areas reserved for other specific purposes.

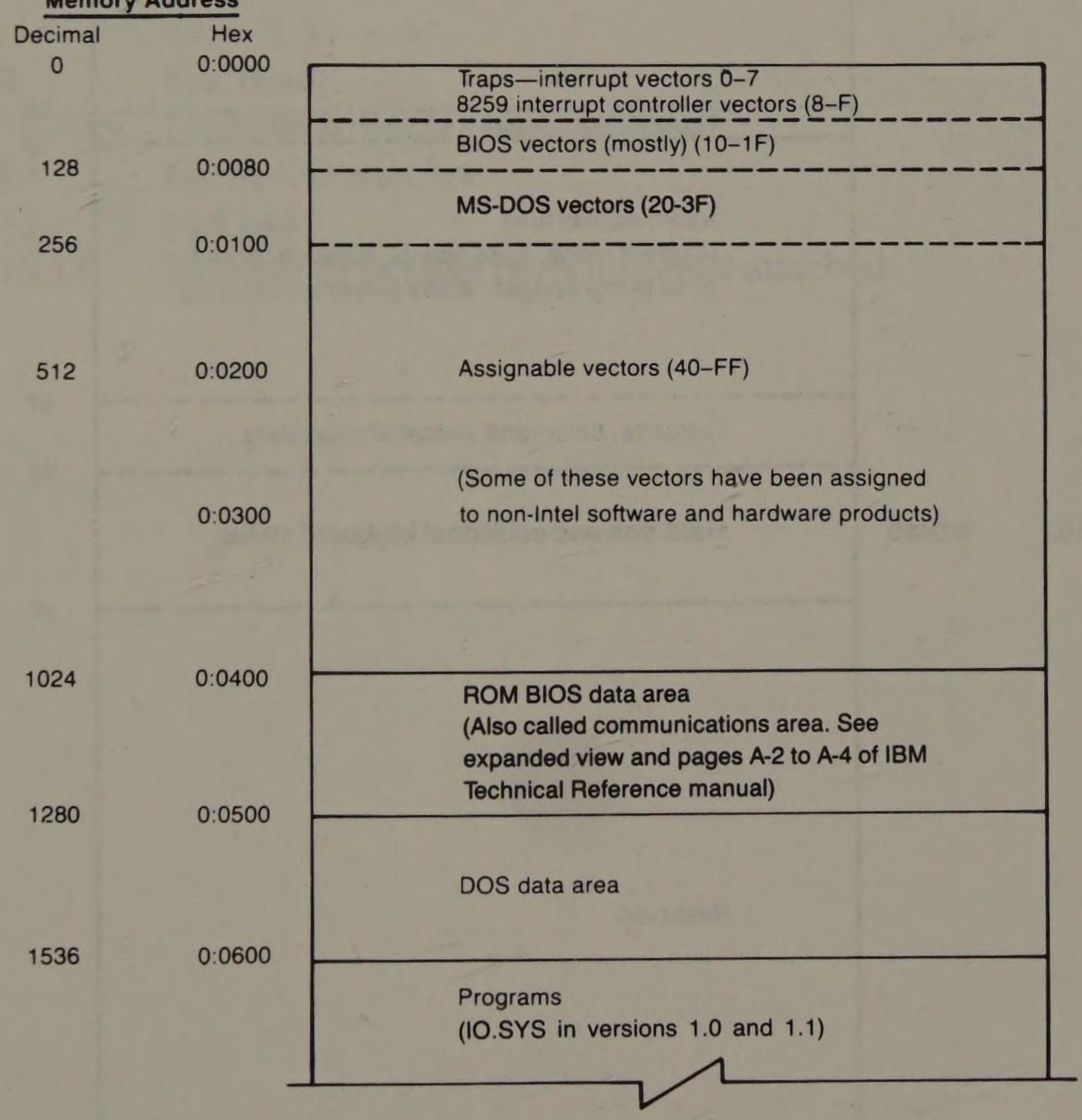
The Whole One-Megabyte Address Space [PCCs]

Hex	Decimal	Contents
00000	0K 16K 32K 48K	Interrupt vectors DOS software Disk and Advanced BASIC (often) Your program(s) and data
10000	64K	
20000	128K	
30000	192K	
40000	256K	
50000	320K	Note there is plenty (at least 98,000 hex or 608K or more than 622,592 bytes) of room for your programs (if you can afford the memory and slot space).
60000	384K	
70000	448K	
80000	512K	
90000	576K	
A0000	640K	This space claimed for fancy graphics capabilities
B0000	704K	Monochrome display's 4K buffer
B8000	736K	Color/graphics display buffers
C0000	768K	
C8000	800K	Hard disk adaptor's ROM
D0000	832K	
E0000	896K	
F0000		Reserved for ROM expansion
F6000	984K	Microsoft (cassette) BASIC (F6000–FDFFF, 32K) — ROM
FE000	1016K	BIOS (FE000–FFFFF, 8K) — ROM

The second map is an expanded low address memory map, showing the allocation of low memory (0:0000 to 0:0400) to (256) interrupt vectors and ROM BIOS and DOS data (communications) areas.

The third map shows an expanded view of the ROM BIOS communications area. This is where BIOS maintains its variable parameters. It is also known as the ROM BIOS data area. For contents of special locations, see the *IBM Technical Reference* (which COMPAQ recommends as the technical documentation for its machine).

Low Memory Map



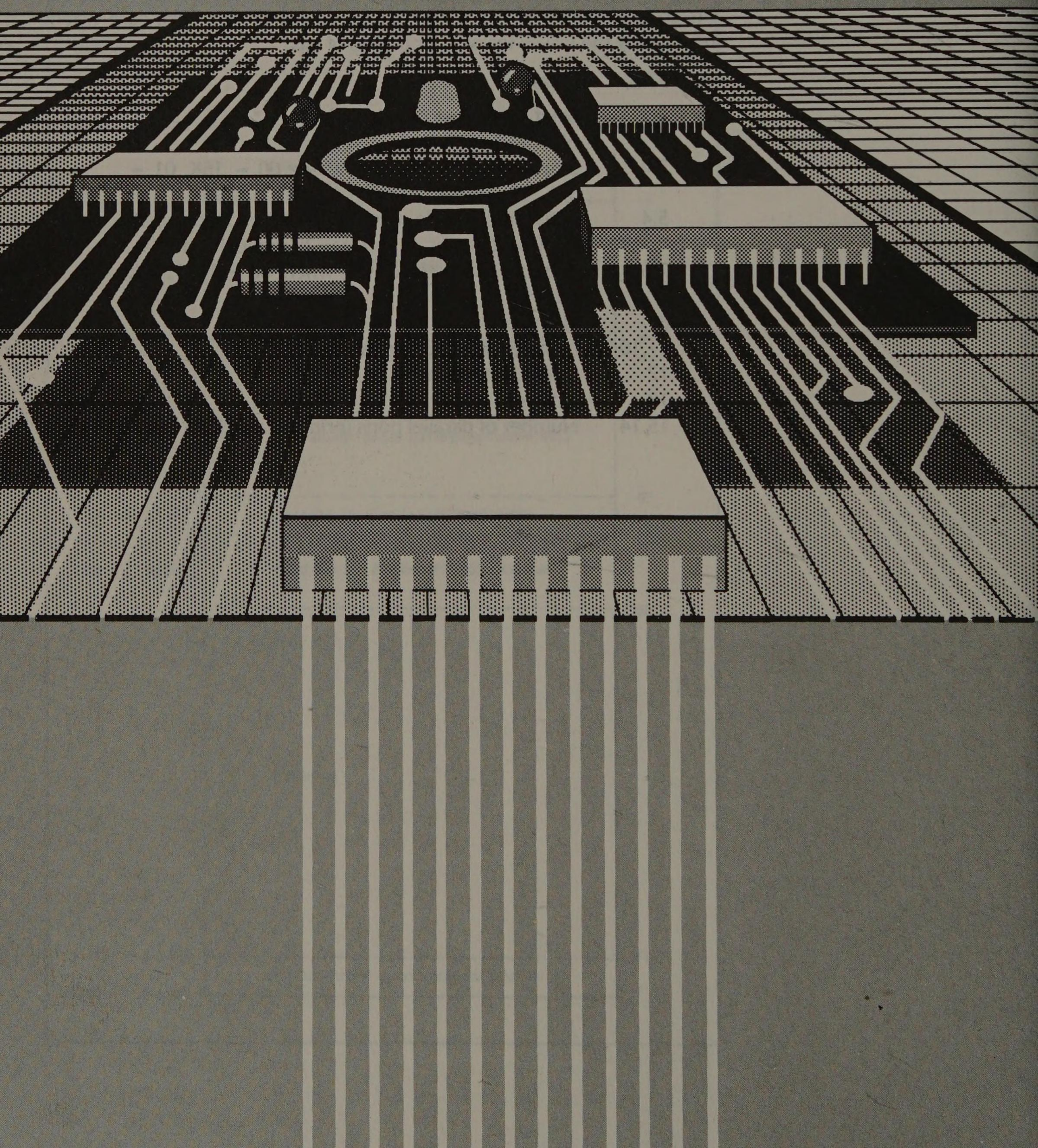
ROM BIOS Communications Area—0:400

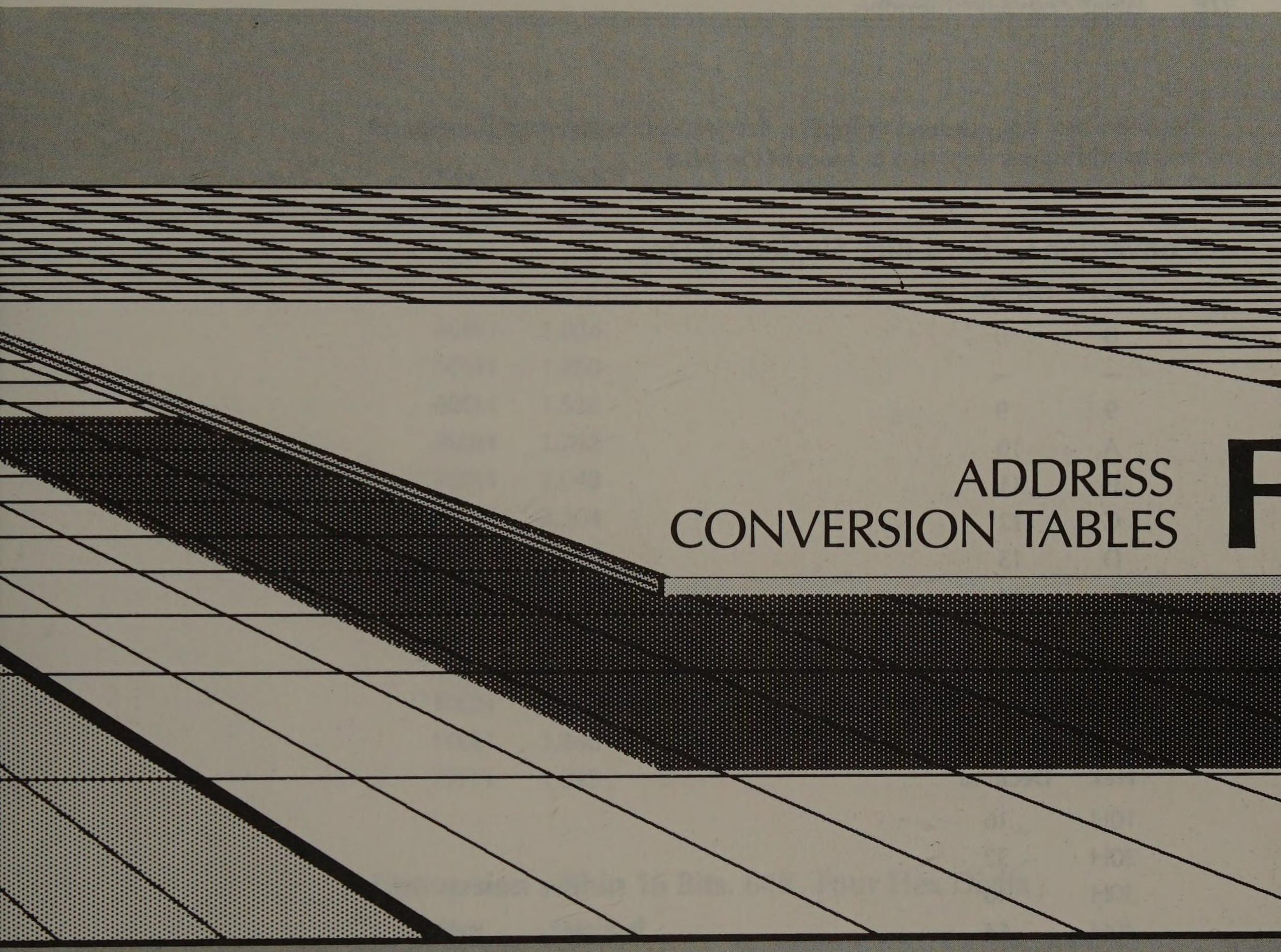
Memory Address			Offset Address
Decimal	Hex		(Hex)
1024	0:0400	Hardware environment information (Printer and RS-232 devices addresses, memory size, etc.)	0
			17
		Keyboard buffer and status	
			3E
		Diskette data area	42
		Video display area (Current mode, color palette; cursor positions of up to eight pages; active page number, etc.)	
			67
		Cassette, timer, and miscellaneous data	
			72
1152	0:0480	**Hard disk and additional keyboard status**	
			8F
		Reserved	
	0:04F0		F0
		Interapplication communications area	
1280	0:0500		

Equipment Flag Values [PCCs]

The meanings of the bits in the equipment flag for the IBM PC, where the bits are numbered from low to high order, are shown. Take care when interpreting the values for pairs or triplets of bits.

Bit(s)	Meaning
0	Set if system has diskette drives
1	Not used
3,2	System board RAM size, where 00 = 16K, 01 = 32K, 10 = 48K, 11 = 64K
5,4	Initial video mode, where 01 = 40 × 25 color, 10 = 80 × 25 color, 11 = 80 × 25 monochrome
7,6	Number of diskette drives, where 00 = 1, 01 = 2, 10 = 3, 11 = 4
8	Not used
11,10,9	Number of RS-232 ports attached
12	Game I/O attached
13	Not used
15,14	Number of parallel ports (printers) attached





ADDRESS CONVERSION TABLES

F

These tables show address conversions for the one-megabyte (1,048,576) address space of the 8086/8088. It takes twenty bits to address one megabyte. The memory is addressed in units of bytes, paragraphs, or segments. A paragraph is sixteen bytes; a segment is 65,536 (64K) bytes. The tables are presented in this format (separate tables for each number of digits in the hex value) in order to give you a quick idea of the correspondence

between hex and decimal numbers. Precise conversion usually requires you to add values from two or more of the tables.

Conversion within 4 Bits, Single Hex Digit

Hex	Decimal
0	0
—	—
9	9
A	10
B	11
C	12
D	13
E	14
F	15

Conversion within 8 Bits, Two Hex Digits

Hex	Decimal
10H	16
20H	32
30H	48
40H	64
50H	80
60H	96
70H	112
80H	128
90H	144
A0H	160
B0H	176
C0H	192
D0H	208
E0H	224
F0H	240
FFH	255

Conversion within 12 Bits, 4K, Three Hex Digits

Hex	Decimal
100H	256
200H	512
300H	768
400H	1,024
500H	1,280
600H	1,536
700H	1,792
800H	2,048
900H	2,304
A00H	2,560
B00H	2,816
C00H	3,072
D00H	3,328
E00H	3,584
F00H	3,840
FFFH	4,095

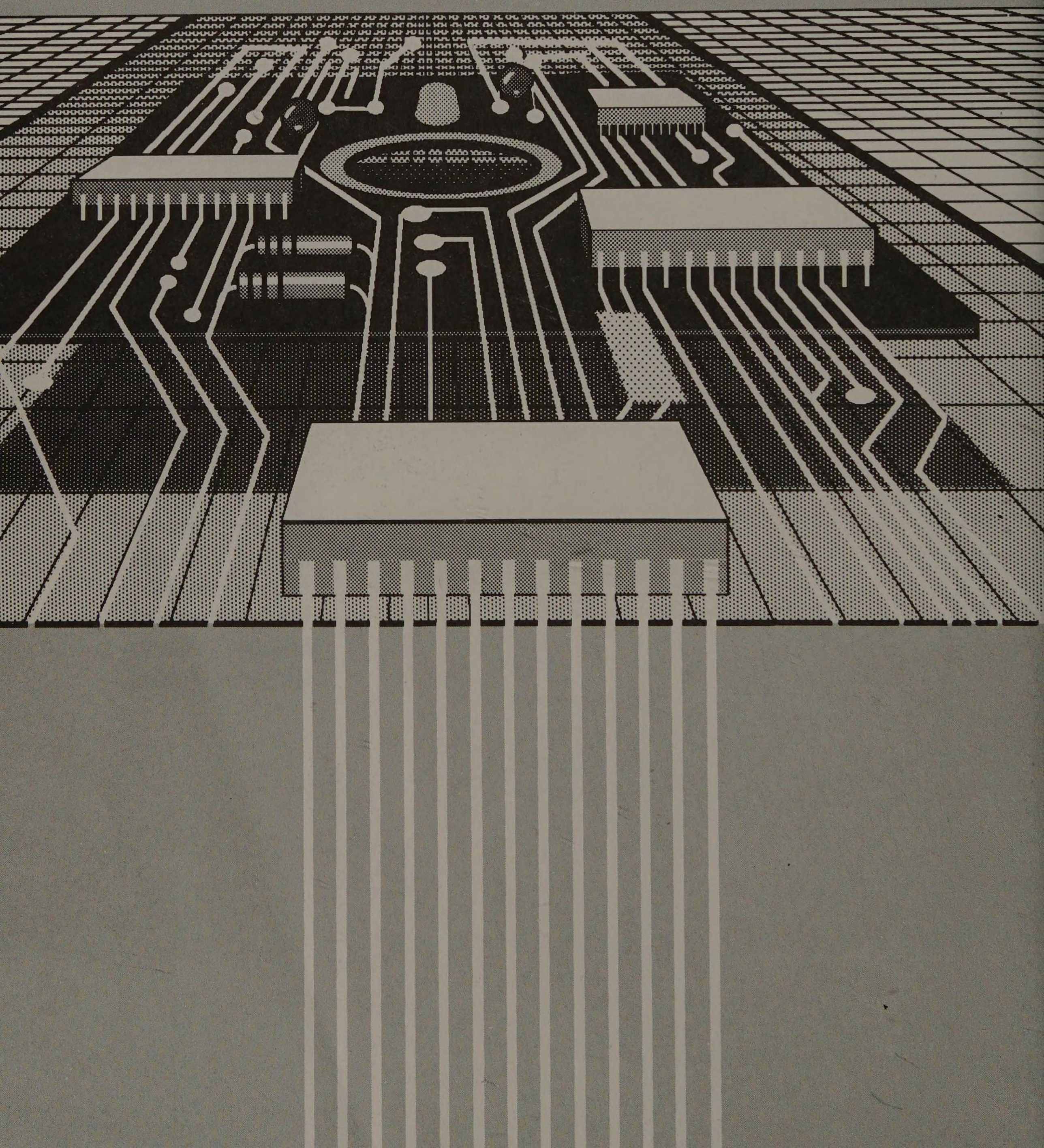
Conversion within 16 Bits, 64K, Four Hex Digits

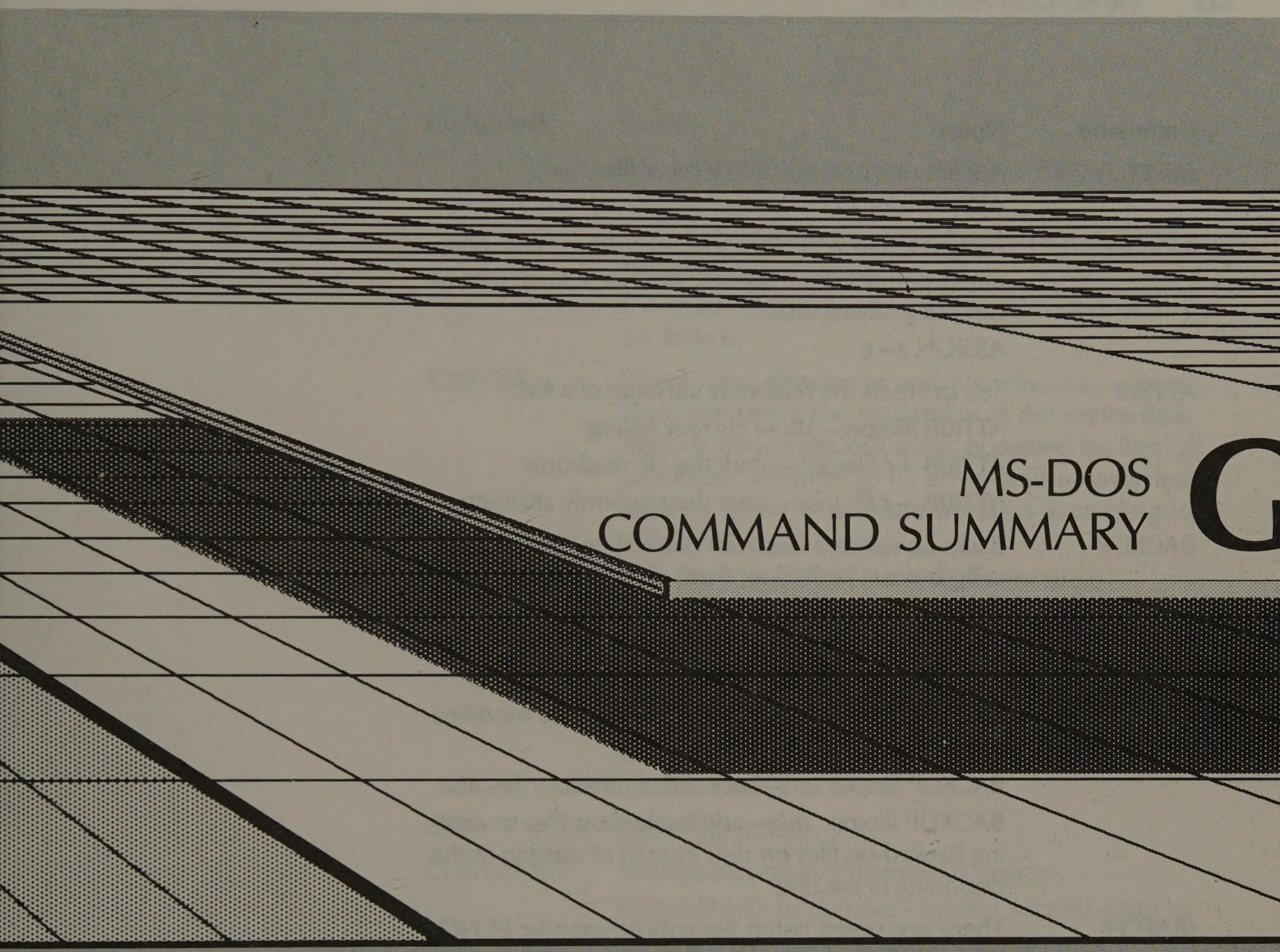
Hex	Decimal
1000H	4,096
2000H	8,192
3000H	12,288
4000H	16,384
5000H	20,480
6000H	24,576
7000H	28,672
8000H	32,768
9000H	36,864
A000H	40,960
B000H	45,056
C000H	49,152

Hex	Decimal
D000H	53,248
E000H	57,344
F000H	61,440
FFFFH	65,535

Conversion within 20 Bits, 1 MB, Five Hex Digits

Hex	Decimal
10000H	65,536
20000H	131,072
30000H	196,608
40000H	262,144
50000H	327,680
60000H	393,216
70000H	458,752
80000H	524,288
90000H	589,824
A0000H	655,360
B0000H	720,896
C0000H	786,432
D0000H	851,968
E0000H	917,504
F0000H	983,040
FFFFFH	1,048,575





MS-DOS COMMAND SUMMARY

G

This appendix provides a summary of the MS-DOS command set. All the commands are well described in the DOS manuals; only a few notes have been added here. Commands not usually part of MS-DOS are marked [un] for uncommon.

All the commands accept a preceding drive and path to tell MS-DOS where they can be found, and all file specifications that define the files that the commands are to operate on can, in version 3, be preceded by a drive and path. The syntax of the command, shown usually by example, follows the description; *filespec* specifies a file name or a group of files. Commands can be typed in either uppercase or lowercase.

Command	Notes
APPLY	Applies one command to a list of files. [un] APPLY *filelist* TYPE %—type all files named in *filelist*.
ASSIGN	Assigns drive A or B to another drive. This command permits drive-specific version 1 programs to run using a hard disk. ASSIGN a=c
ATTRIB	Sets or resets the read-only attribute of a file. ATTRIB *filespec*—show current setting ATTRIB +r *filespec*—mark the file read-only ATTRIB –r *filespec*—clear the read-only attribute
BACKUP	Backs up selected files from one disk to another, typically, but not limited to, from a hard disk to floppy disks. BACKUP *filespec* d:/m—back up *filespec* files modified since last backup onto media in disk drive D BACKUP *filespec* d:/d:*date*—back up files modified since *date* BACKUP *filespec* d:/s—back up subdirectory files also BACKUP *filespec* d:/a—add backed-up files to existing backed-up files on disk instead of starting at the beginning of the disk
(BATCH)	There are seven batch file subcommands: ECHO, FOR, GOTO, IF, PAUSE, REM, and SHIFT. BATCH itself is not a command. See Chapter 14, "Batch Operations."
BREAK	Controls and displays the level of Ctrl-Break detection. BREAK—show current level (extra break checking on or off) BREAK on—turn on extended break checking BREAK off—turn off extended break checking
CHDIR	Changes or displays the current directory for the specified drive (uses current drive if none is specified). May be abbreviated to CD. CD—show current directory of current (default) drive CD b:—show current directory of drive B CD\—make the current directory the root directory

Command	Notes
	CD ..—make the current directory the preceding level in the tree
	CD clothes\brown—make the current directory become clothes\brown
	CD b:assess\stocks—change the current directory on drive B
CHKDSK	Checks the directories and file allocation tables. Does not verify the readability of the entire disk. Reports noncontiguous areas occupied by files. If requested, attempts to fix errors and recover lost clusters. Reports memory installed according to switch settings.
	CHKDSK c:—provide status report for drive C
	CHKDSK c:/f—provide status report for drive C and fix errors
	CHKDSK b:*.*—provide status report and list non-contiguous files
	CHKDSK b:/v—provide status report and list all files on disk B
CLS	Clears the screen.
COMMAND	Invokes secondary command processor.
	COMMAND /c *doit.bat*—invoke second copy of command processor and execute *doit.bat*
	COMMAND—invoke second copy of command processor; use EXIT to return
	COMMAND/p—invoke second copy and make it permanent
COMP	[PCCs] Compares individual files or sets of files. Useful for checking that two files are identical, or that a few (up to ten) locations contain different values. This command is not useful if one file has one or more extra bytes, except to report the location of the first extra byte. (See also FILCOMP.)
CONFIGURE	Configures the baud rate, parity, number of bits per word, and number of stop bits of the serial port. Permits redirection of parallel port output to the serial port. Also seen as CONFIG. See MODE for more information.

Command	Notes
COPY	Copies one or more files from one disk to another. Concatenates files on request. Also copies files from one device to another, for instance from console to disk or disk to printer. Because of the file-name wild-card (global) characters (? and *), and the /A (ASCII), /B (binary) and /V (verify) options, this is a powerful, flexible command. Do read the manual for this one. The COPY *.* b: command is useful for compressing randomly distributed files onto a new disk. COPY *filespec1 filespec2*—copy from *filespec1* to *filespec2* COPY *filespec1 filespec2*/v—verify copy after writing it COPY CON *filespec*—copy characters from the keyboard into *filespec* until a Ctrl-Z is entered COPY *filespecs*/a *filespec*—stop concatenation at Ctrl-Z (end of file) character in each of *filespecs* COPY *filespecs filespec*/a—add Ctrl-Z at the end of *filespec* COPY *filespecs*/b *filespec*—concatenate entire files into *filespec* COPY *filespecs filespec*/b—do not add Ctrl-Z at the end of *filespec*
CTTY	Changes standard input and standard output to an auxiliary console. CTTY COM1—drive MS-DOS from COM1 CTTY CON—return to normal operations, i.e., drive MS-DOS from the keyboard
DATE	Displays the date and asks for a new one.
DEL	Deletes a file or files. See ERASE. DEL *filespec*
DIR	Lists all or some of the files in a directory. The filename, extension, size in bytes, and date and time of last alteration are shown. DIR b:/w—show, five filenames to a line, the files in the root directory of B DIR c:\pilots—show, one file to a line, files in the directory pilots

Command	Notes
	DIR/p—show files in the current directory, pausing each screenful
DISKCOMP	(Sometimes DSKCOMP) Compares entire diskettes. Useful for verifying the readability of the whole diskette.
DISKCOPY	(Sometimes DSKCOPY) Copies entire diskettes, byte for byte. Use DISKCOMP afterwards if you need certainty that the new copy is perfect. DISKCOPY a: b:—copy disk in drive A to disk in drive B DISKCOPY a: b:/1—copy the first side only
DISKINIT	Creates a configured MS-DOS system diskette.
ERASE	Erases (forever, unless you have special recovery software) one or more files. Use with care. ERASE *filespec*—erase file specified
EXE2BIN	Under the right conditions, converts an .EXE file into a .BIN file. The resultant .BIN file may have the format of a .COM file and only require a filename extension change to become a .COM file and to be recognized as such. The .BIN file may require loading at an absolute location.
EXEFIX	Modifies the header in some .EXE files. [un]
EXIT	Exits the current command processor and returns to the previous COMMAND processor.
FILCOMP	A file comparison utility; a more capable alternative to COMP. See the user's manual for manufacturer-specific alternatives.
FDISK	A menu-driven hard-disk preparation program.
FIND	A filter that searches the data stream for a text string. FIND *"string"filespec*—show lines in the specified file that contain *string* FIND/n *"string"filespec*—show lines, with line numbers, in the specified file that contain *string* FIND/v*"string"filespec*—show lines that don't contain the *string*

Command	Notes
	FIND/c"*string*"*filespec*—count lines containing the *string* DIR¦FIND "<DIR>"—show all subdirectories
FORMAT	Formats disks. Use with care. FORMAT totally erases any previous information from a disk. It will erase drive C late at night when you are tired. Rename this command and create FORMAT.BAT that offers warning prompts before destroying data. Read the user's guide for more options and information. (The DISKCOPY command will format unformatted diskettes for you.) FORMAT b:—format disk in drive B FORMAT b:/s—format disk in B and add MS-DOS to it FORMAT b:/v—format disk in B and ask for a volume label FORMAT b:/8—format 8 sector-per-track disk
GRAFTABL	Loads above-127 ASCII set of graphics characters for the color/graphics adaptor. GRAPHTBL—load character set into memory (use only once per boot)
GRAPHICS	When invoked, Shift-PrtSc prints the text or graphics image shown on the color/graphics monitor. Will accept different printer types. GRAPHICS—invoke ability to print color/graphics images GRAPHICS/r—print black where white would normally be on the printed image, i.e., don't reverse black and white between screen and paper
JOIN	Joins a whole drive to a subdirectory of another drive. Useful for using hard disks with programs that only understand diskettes. JOIN c: a:\harddisk—make drive C files accessible via the subdirectory harddisk on drive A JOIN c: /d—disconnect the join
KEYBxx	Loads code that supports non-English keyboards. KEYBGR—install the German keyboard support
LABEL	Manipulates a disk's volume label.

Command	Notes
	LABEL b:circuits—make the volume label for the disk in drive B be circuits
	LABEL b:—delete the volume label from the disk in drive B
MKDIR	Makes a directory. May be abbreviated to MD.
	MD \tools—add, just below the root, the directory \tools
MODE	[PCCs] A multifunction command. See Chapter 13, "Configuring the System," for more information.
	MODE lpt1:132,6,p—set 132 columns per line, 6 lines per inch, and retry on printer timeout
	MODE bw40—set 40 characters per line for color/graphics-driven display
	MODE mono—return to monochrome monitor, 80 characters per line
	MODE r,t—align image on screen by shifting it right (or left) and provide a test pattern
	MODE COM1:2400,e,7,1—set baud rate to 2400, even parity, 8 bits per word, and 1 stop bit for the COM1 serial port
	MODE COM1:2400,e,7,1,p—set baud rate to 2400, even parity, 8 bits per word, and 1 stop bit, with retry on printer timeout
	MODE LPT1=COM2—redirect parallel printer output to a serial port
MORE	A filter that displays data one screen at a time.
PATH	Defines search paths for programs not in the current directory.
	PATH d:\;d:\tools;e:\—look for commands first on drive D's root directory, then in the \tools subdirectory of drive D, then in drive E's root directory
PAUSE	Used in a batch file to display a message, then wait for any keystroke.
PRINT	Prints a queue of files as a background task.
	PRINT /d:LPT2/b:1024/q:20—set up the background print task for LPT2, with a 1024-byte buffer and a queue of 20 files
	PRINT freddy.lst—add freddy.lst to the print queue

Command	Notes
	PRINT fairway.lst/c—remove the request to print fairway.lst from the queue
	PRINT /t—cancel all printing, remove all files from the queue
	PRINT /s:10/u:2/m:3—increase timeslice for print task from default 8 ticks to 10; wait for 2 ticks for printer ready instead of the usual 1 tick; increase to 3, from the default value of 2, the number of ticks the background print task can have to print characters
PROMPT	Changes the appearance of the MS-DOS prompt. See Chapter 13, "Configuring the System," for more details.
	PROMPT $p $g—make the MS-DOS prompt show the current directory followed by >
RECOVER	Recovers damaged files.
	RECOVER *filespec*—depending on *filespec,* attempt recovery of a single file, a set of files, or a whole disk
REM	Displays a message from a batch file.
RENAME	Changes the name of a file.
	RENAME bill.dat william.dat—rename the file bill.dat to william.dat
RESTORE	Restores files backed-up by BACKUP.
	RESTORE a: c:*.* /s—restore all files from disk in drive A to disk in drive C, including files in subdirectories; ultimate restoration
	RESTORE a: c:\tools /p—restore to c:\tools, prompting for read-only files and for files changed since last backup
RMDIR	Removes a directory. May be abbreviated to RD.
SEARCH	Locates files in a tree structure. [un]
SELECT	Creates configured MS-DOS system diskette.
SET	Inserts strings into the command processor's environment.
	SET vegetable=broccoli—put the string vegetable=broccoli into the environment
	SET vegetable=—delete the string vegetable=broccoli from the environment

Command	Notes
SHARE	Loads file sharing for network operations. SHARE /f:3000/l:30—allow for 50% more than default (2048 bytes) file sharing status buffer size, and allow for 30, instead of the usual 20, locks
SORT	A filter that sorts text data. SORT /r < *textfile*—sort lines in *textfile* in reverse order SORT /+6 < *textfile*—sort lines in *textfile* starting at column 6
SUBST	Uses another drive specifier to refer to another drive or a subdirectory. This allows you to use some programs that don't recognize paths. SUBST b: e:\tools—make it possible to use b: to refer to e:\tools SUBST b: /d—delete the previous substitution
SYS	Puts the DOS operating system files on a diskette that has been formatted to allocate space for these files. SYS b:—transfer the DOS system to the diskette in drive B
TIME	Displays, and permits change of, the time of day.
TREE	Displays on the screen the tree of directories on a disk. TREE b:—show the directories on the disk in drive B TREE b:/f—show the directories and files in B
TYPE	Displays the contents of a single file.
VER	Displays the MS-DOS version number.
VERIFY	Sets and displays the read-after-write verify state. VERIFY on—turn on read-after-write verify
VOL	Displays the volume name of a disk.

INDEX

#, 91, 167
$, 205
$e, 77
@, 191
%, 220
%%, 230
%PIPE, 160, 191
*, 45, 156, 166, 230, 275
$$$, 166, 174, 191
/S option, 55
<, 161
>, 159–161, 276
>>, 159
?, 45, 156, 230, 275
\, 20–21, 180, 183, 186
|, 159–161

A

Abbreviating commands, 221
Abort batch file operations, 219–220
Absolute sector, 54
Access code, 139
Access modes, 22–23
Active page, 94–95
Adaptor boards, 242
Adaptors
  color/graphics, 85–87, 89, 96, 100, 202, 235–237, 242
  monochrome, 85–86, 202, 234
Address conversions, 6, 315–318
Addressing memory, 5
AH register, 7, 41, 61
AL register, 29, 41, 44–45, 49, 139, 296
All-Points-Addressable mode (APA), 236–237
Alphanumeric mode, 89, 93, 238
ANSI (American National Standards Institute), 201
ANSI.SYS, 75–76, 90, 201
Append command, 166, 171
APPLY command, 275–276, 322
ASCIIZ path, 21
Assemble command, 260, 264, 269

ASSIGN command, 212, 322
Asynchronous Communications Adaptor. *See* Serial ports
Asynchronous Communications Support, 250
Asynchronous port, 11
ATTRIB command, 322
Attributes, character, 85–88, 92, 97, 100, 239
Audio feedback, 203
AUTOEXEC.BAT, 139, 198, 211, 218, 221
Automatic program execution, 220

B

Backslash, 20–21, 180, 183, 186
Backspace, 84
BACKUP command, 207, 322
BADTRACK, 191
BAK files, 174, 191
BASIC, 152, 280, 292
BATCH commands, 322
Batch file processor, 219, 222, 224
Batch files, 217–218, 222
  aborting, 219–220
  calling, 223
  chaining, 227–228
  creating, 220
  sample, 225
  terminating, 219–220
Baud rate, 105–106
Bidirectional communication, 108
Binary files, 173
Bits, 267
Blinking bit, 86
Blocks, 34, 178
Break character, 10, 66–67, 166, 198, 219, 296
Break checking, 67
BREAK command, 198–199, 203, 259, 322
Break-detection control, 296
Breakpoints, 259
Budgets, 181
Buffered keyboard input, 69
BUFFERS parameter, 199
Byte registers, 7
Bytes, 186

C

Cabling, 247–249
Calling functions, 8
Campbell, Joe, 109
CapsLock key, 71
Carry flag, 24, 60
Cartesian coordinates, 91
CD command, 181, 322
Character attributes, 85–88, 92, 97, 100, 239
Character device I/O, 10–11
Character I/O, 94, 97–98, 296
Character strings, 84, 90
Characters
  blinking, 239
  changing meaning of, 73
  colored, 87
  creating, 237
  displaying, 234, 236
  getting, 66, 79
  locating, 100
  pending, 67–68
  printing, 109–110
  receiving, 10, 107
  sending, 10, 84, 96, 106
  single, 78
  translating, 74
  wild-card, 156, 209, 275
CHDIR command, 181, 322
CHKDSK command, 56, 129, 157, 188, 221, 268, 323
Clock, 118–119, 218
Close function, 44
CLS command, 323
Clusters, 58–59, 178, 193
Color, 87
  background, 239
  capabilities, 235
  selection, 237–241
  setting palette, 98–99
  tables listing, 234

Color/graphics adaptor, 87, 89, 96, 100, 202, 234–237, 242
Color monitors, 240–241
Command line creation, 127
Command name string, 128–129
Command processor, 16, 65, 213, 218–219
COMMAND/C operation, 229
COMMAND.COM, 16, 55–56, 65, 129, 213, 323
Communications capabilities, 246
Communications software, 249–250
COMP command, 323
COMPAQ, 56–57, 119, 268
Compare command, 258, 269
Compatibility, 273–274, 278–279, 281
CONFIG command, 202
CONFIG.SYS, 198–200
Configuration, 197–202
CONFIGURE command, 103–104, 246, 275, 323
COPY command, 75–76, 153, 156, 172, 193, 220–221, 324
COPYALL key, 151
Country-dependent information, 11, 199, 297
COUNTRY parameter, 199
Crashes, 174, 185, 207
Create function, 24, 46
CRT controller chips, 93
Ctrl-Break, 78, 198, 296
Ctrl-C, 166, 198, 296
Ctrl key, 71
Ctrl-NumLock, 154
Ctrl-Q, 154
Ctrl-S, 154, 156
Ctrl-V, 173
Ctrl-Z, 156, 173
CTTY command, 212, 324
Current block field, 40
Current record field, 40
Cursor control, 89, 91–92, 95–96

D

Data
  access, 19, 33
  compatibility, 279
  organization, 187
  reading, 12, 298
  recovering, 208
  transferring, 27, 38, 40–41
  writing, 12, 298
Date, 11, 33, 113–115, 218, 297, 324
DATE command, 218, 324
DEBUG command, 76–77, 152, 253–266, 269
Debugging, 253–266
Del command, 268, 324
Del key, 72
Deny write, 140
Description files, 229
Device driver, 201
Devices
  investigating status, 296
  names, 145, 155, 191
  redirection, 142
  remote or local, 140–141
  status report, 91
Device I/O, 10–11, 29, 296
DEVICE parameter, 201
Diablo 630, 247
DIR command, 156, 324
Direct console I/O function, 85
Directories
  accessing, 12–13, 298
  changing, 31
  creating and deleting, 30–31
  finding files in, 30
  initializing, 46
  manipulating, 13, 298
  network, 143
  searching, 45–46
  tree-structured, 20–21, 177–184, 209
  viewing, 222
Disk drives, 61–62
DISKCOMP command, 325
DISKCOPY command, 157, 274, 277, 325
DISKINIT command, 325
Disks
  accessing, 12, 59–60, 297, 299

Disks (continued)
bootstrap, 15–16
capacities, 56–57
compatibilities, 281
copying, 274
errors, 60–61, 155
identifying, 178
managing, 3, 16
maps, 55
parameters, 32
recovering, 209
sector numbering, 54–55
space, 48, 52, 58
tracks, 53
transferring, 48, 60, 298
Display modes, 234
DO command, 223
Documentation, 276
Dollar sign, 85
DOS manual, 155
Dots, 239, 241
Double-sided disk, 56–57
DSKCOMP command, 325
DSKCOPY command, 157, 274, 277, 325
Dummy arguments, 220, 222, 224, 229
Dump command, 258, 269

E

ECHO command, 67, 223–225, 230
Edit command, 168
Editing, 163–164, 166, 168
a binary file, 173
capabilities, 69
a command, 151
ending the session, 173
in-line, 151–154
keys for, 151, 169
short lines, 170
EDLIN, 76, 163, 167–169, 173–175
commands, 151–152, 165–166
creating batch files with, 220
End command, 166
Enter command, 269
Envelope function, 69
Environment strings, 127, 229
Equipment flag, 267, 313
Erase command, 268, 325
Erasure sequences, 92
Error classes, 133–134
Error codes, 12, 27, 130–133, 297
Errors, 24, 60–61, 141
disk, 154–155
extended, 130, 142
reporting, 123, 129–130
Esc [, 90–91
Esc key, 71
Escape character, 77, 205
EXE2BIN command, 325
EXEC function, 104, 107
Execute function, 125–126
EXEFIX command, 325
EXIT command, 325
Expansion boards, 280
Extended codes, 72
Extended screen control, 89–90

F

FCBS parameter, 199–200
FDISK command, 206, 325
FILCOMP command, 277, 325
File-access functions, 25
File allocation table (FAT), 58–59, 268
File-comparison program, 275
File control blocks (FCBs), 13, 34–40, 45–48, 127–128, 200, 299
File handles, 12, 19, 21–22, 27, 29, 33
File I/O, 296
File management capabilities, 3, 16
Files
accessing, 12–13, 22–24, 35–41, 141, 277, 298–299
attributes of, 22, 32
binary, 173
capacities, 186
closing, 26, 44
copying, 156
creating, 24, 26, 44, 46, 166, 298
date and time, 33
deleting, 27, 44, 47, 298

Files (continued)
describing, 229
displaying names, 156
finding, 44–45, 156, 298
finding in directory, 12, 14, 30, 299
handling, 12, 19, 21–22, 27, 29, 33, 139, 277, 298
in batches, 217
initializing, 46
joining, 211
locating, 20
locking, 142–144
naming, 161, 189–193, 219, 275
opening, 26, 38–39, 139, 200, 298
organizing, 177–178, 186–187
placing clusters, 193
pointing handle to another file, 29
printing, 157
queuing, 157, 209, 211
reading from, 27, 40, 298
reading to disks under CP/M, 275
recovering, 268
remote or local, 140
renaming, 27, 46, 298
retrieving, 208
sequential, 40–41
setting address, 298
sharing, 138, 142, 199, 214
size, 14, 20, 47, 188, 299
systems, 20, 275
temporary, 26
tree-structured, 20, 177–183, 209
whole, 299
writing to, 28, 40, 298
FILES parameter, 200
Fill command, 260, 269
Filters, 158, 160
FIND command, 160, 325
Flags
carry, 24, 60
equipment, 267, 313
inheritance, 140
FONT command, 275
FOR command, 223, 227
FORMAT command, 15, 193, 207, 275, 277, 326
Function calls, 3, 8, 277
affecting screens, 84
descriptions, 9–10
file access, 12
in numerical order, 301–303
register settings, 38
summaries, 295
Function keys, 71–72

G

Get machine name, 145
Go command, 258–259, 263, 269
GOTO command, 223–224
GRAFTABL command, 326
Graphics, 87, 90–92, 234
annotating image, 236
capabilities, 235
color options, 240
modes, 89, 93–99
output, 98
resolution, 239–240
Graphics adaptor. *See* Color/graphics adaptor
GRAPHICS command, 158, 211, 326

H

Hard disks, 53–56, 62, 198, 206
Hardware, 255, 279
Hex command, 269
Home key, 72
Horizontal scan line, 96

I

IBM PC-compatible computers, 278–279
IBM Technical Reference, 106, 109
IBMBIO.COM, 55
IBMDOS.COM, 55
IF command, 223
Image alignment, 202
In-line edit, 151–154, 169
Information retrieval services, 249
Inheritance flag, 140
Initialization, 46

Input buffer, 69
Input command, 269
Input/output (I/O), 104
  capabilities, 16
  control, 10, 29, 296
  redirection, 108–109, 158–159, 296
  support, 4
Insert command, 166, 168
Interrupt 21H, 9–10, 23, 35, 46, 104
Interrupt vectors, 7–8, 67, 266, 300, 305–307
Interval-timer functions, 113
IO.SYS, 16, 55, 268
IOCTL, 140

J

JMP instructions, 268
JOIN command, 211, 326

K

Key codes, 71
Key reassignment, 70, 75–78, 154
Keyboard, 11, 71, 150–151, 297
  accessing, 66, 68–69
  buffered input, 69–70
  compatibility, 281
  interrupts, 72, 153
  layout, 71
  reassignment of keys, 70, 75–78, 154
  status, 67–68
KEYBxx command, 326

L

Label, 224
LABEL command, 326
LASTDRIVE command, 143, 200
Light pen, 93, 99
Line length, 152
Line numbers, 164–165, 167
List command, 167
Load-and-execute command, 126
Load command, 269
Locus, 134
Logical sector, 54

M

Macro assembler, 276
MAKE command, 275–276
MD command, 183, 327
Memory
  addressing, 5
  allocating, 124
  changing size, 125
  deallocating, 124
  expansion board, 280
  management, 14, 124, 126, 299
  maps, 309–313
  paragraphs, 124–125
  returning to pool, 126
Memory-resident programs, 123
Meta-strings, 204–205
Microsoft, 273, 276
MKDIR command, 183, 327
Mnemonics, 91, 193
MODE command, 202–203, 246, 250, 275, 277, 327
Mode-setting sequences, 92, 107
Modems, 245, 249–250
Monitors
  choosing, 241
  color, 240–241
  drivers, 88
  mixing, 242
Monochrome adaptor, 86, 202, 234
Monochrome display, 86, 202, 234–235, 242
MORE command, 160, 327
Move command, 173, 269
MS-DOS
  capacities, 150
  commands, 274, 321
  communications capabilities, 246
  compatibility, 273–274
  components, 15
  difference from PC-DOS, 277–278
  fundamentals, 2–3
  versions, 283–293

MSDOS.SYS, 16, 55
Multiple-record random write, 43
Multitasking, 125, 128, 218

N

Name command, 265, 269
Network directories, 143
Network operations, 14, 129–130, 138–145, 200, 213–214, 300
Network paths, 186
Network redirection, 142–143
Numeric keys, 72
NumLock key, 72

O

Open function, 38–39, 140
Open mode, 23, 139
Output command, 270

P

Page command, 167–168
Parallel ports, 11, 103–104, 297
Parallel printer control, 109
Parameter block, 126–127, 129
Parameters protocol, 250
Parse function, 45, 48–49, 298
Password, 143
Patching, 261, 267
Path, 20–21, 180
PATH command, 185, 327
PAUSE command, 222, 327
PC-DOS, 277–278, 286
Pels, 239, 241
PgDn key, 72
PgUp key, 72
Pipes, 158–161
Pixels, 239, 241
Plus-sign key, 72
Points, 241
Ports
  parallel, 103–104, 297
  serial, 103–107, 245–249, 297
PRINT command, 157–158, 160, 209–211, 327
Print spooling, 157
Printers, 11, 110, 297
  characteristics, 248
  interface, 109, 245–246
  mode selection, 203
  output to parallel, 109, 246
  setup strings, 144
Printing, 152, 160
  files, 157
  graphics, 211
  queue, 157, 209, 211
  screen image, 153
  speed, 211
Proceed command, 258, 260, 270
Process handling, 14, 300
Program segment prefix (PSP), 8
Programmers, 2–4
Programs
  altering, 266
  creating with DEBUG, 263
  executing, 220
  portability, 4
  suspending, 68
  terminating, 300
ProKey program, 123
PROMPT command, 77, 204–205, 276, 328
Prompt message, 276
Protocol parameters, 250

Q

Question-mark option, 170
Queue, print, 157, 209, 211
Quit command, 166, 270

R

Random-file-access functions, 40–42
Random read function, 42
Random record field, 41
RDCPM command, 275
Read-after-write verify operation, 32
Read/write pointer, 28
Record-size field, 40
Records, 13, 34, 42–43, 299
RECOVER command, 208–209, 328

Recovery operations, 208–209, 268, 328
Redirection, 142–144, 158–159, 247
Register command, 258, 260, 270
Register settings, 38
Relative sector, 54
REM command, 222, 328
RENAME command, 221, 328
Replace command, 170–171
RESTORE command, 208, 328
Rewrites, 73–74, 188
RM command, 183, 328
RMDIR command, 183, 328
ROM BIOS, 3–5, 93, 96–98
  chip, 280
  communications area, 267
  display output routines, 267
  keyboard interrupt, 72
  modes, 235
Root directory, 21, 178–179
RS-232 standard, 108–109, 246–248

S

Scan codes, 70–72
Screens, 84, 96
  control, 89–92
  print image, 158
Scrolling, 93, 98
Search command, 170–171, 270, 328
Searching, 45–46, 170–171, 270, 299, 328
Segment registers, 6
SELECT command, 328
Sequential records, 13, 299
Serial communication, 11, 108
Serial interface, 246
Serial ports, 103, 245–246, 297
  cabling, 248–249
  initialization, 105
  management, 104
  output to, 105
  problems with, 247
SET command, 206, 229, 328
SHARE command, 138, 329
Sharing mode, 138–140
Sharing retry count, 141
SHELL parameter, 200
SHIFT command, 224–225
Sidekick, 123
Single-character input functions, 78
Single-record random write, 43
Software development, 275–276
SORT command, 160, 329
Spelling checker, 161
Standard input device, 79
Status checking, 27
Strings, 85, 90, 127, 159, 204, 206, 229
Subdirectories, 179–180
SUBST command, 212, 329
Suggested action code, 134
Syntax, command, 155
SYS command, 329
SYSCOPY, 277

T

Tesla, Nikola, 257
Text mode, 89, 93, 238
TI Professional, 56
Time, 33, 113, 297, 329
  computing differences, 117, 119
  formats, 115–119
  location of cells, 119
  overflow, 118–120
TIME command, 329
Time-interval management, 120
Time-of-day function, 113, 118
Timekeeping routines, 117
Timer, 120
Toggles, 153
Trace command, 258–259, 270
Tracks, 53
Transfer commands, 171–172
TREE command, 184, 329
Tree-structured directories, 20–21, 177–184, 209
TYPE command, 329

U

Unassemble command, 265, 270
User, 2–3, 149
Utility functions, 3, 13, 31, 48, 298

V

Vector access, 15, 300
Verify, 15, 32, 300, 329
VERIFY command, 329
Verify-read after write, 15
Version number access, 15, 300
Video I/O, 93
Video options, 87–88
VOL command, 329

W

Wait function, 68, 128
Wait/retry sequence, 141
Whole-file access, 298
Wild-card characters, 156, 209, 275
Wrapping, 152
Write command, 166, 171, 260, 265, 270

Z

Zero flag, 79

Selections from The SYBEX Library

Computer Specific

CP/M SYSTEMS

THE CP/M HANDBOOK
by Rodnay Zaks
320 pp., illustr., Ref 048-2
An indispensable reference and guide to CP/M – complete in reference form.

MASTERING CP/M
by Alan Miller
398 pp., illustr., Ref. 068-7
For advanced CP/M users or systems programmers who want maximum use of the CP/M operating system: this book takes up where the CP/M Handbook leaves off.

THE CP/M PLUS HANDBOOK
by Alan Miller
250 pp., illustr., Ref. 158-6
This guide is easy for beginners to understand, yet contains valuable information for advanced users of CP/M Plus.

IBM PC AND COMPATIBLES

OPERATING THE IBM PC NETWORKS
Token Ring and Broadband
by Paul Berry
363 pp., illustr., Ref. 307-4
This tells you how to plan, install, and use either the Token Ring Network or the PC Network. Focusing on the hardware-independent PCN software, this book gives readers who need to plan, set-up, operate, and administrate such networks the head start they need to see their way clearly right from the beginning.

THE ABC'S OF THE IBM PC
by Joan Lasselle and Carol Ramsay
143 pp., illustr., Ref. 102-0
This book will take you through the first crucial steps in learning to use the IBM PC.

THE IBM PC-DOS HANDBOOK
by Richard Allen King
296 pp., Ref. 103-9
Explains the PC disk operating system. Get the most out of your PC by adapting its capabilities to your specific needs.

BUSINESS GRAPHICS FOR THE IBM PC
by Nelson Ford
259 pp., illustr. Ref. 124-1
Ready-to-run programs for creating line graphs, multiple bar graphs, pie charts and more. An ideal way to use your PC's business capabilities!

THE IBM PC CONNECTION
by James Coffron
264 pp., illustr., Ref. 127-6
Teaches elementary interfacing and BASIC programming of the IBM PC for connection to external devices and household appliances.

DATA FILE PROGRAMMING ON YOUR IBM PC
by Alan Simpson
219 pp., illustr., Ref. 146-2
This book provides instructions and examples for managing data files in BASIC Programming. Design and development are extensively discussed.

ESSENTIAL PC-DOS
by Myril and Susan Shaw
300 pp., illustr., Ref. 176-4
Whether you work with the IBM PC, XT, PC jr. or the portable PC, this book will be

invaluable both for learning PC DOS and for later reference.

Software Specific

SPREADSHEETS

MASTERING SUPERCALC 3
by Greg Harvey
300 pp., illustr., Ref. 312-0
Featuring Version 2.1, this title offers full coverage of all the sophisticated features of this third generation spreadsheet, including spreadsheet, graphics, database and advanced techniques.

DOING BUSINESS WITH MULTIPLAN
by Richard Allen King and Stanley R. Trost
250 pp., illustr., Ref. 148-9
This book will show you how using Multiplan can be nearly as easy as learning to use a pocket calculator. It presents a collection of templates for business applications.

MULTIPLAN ON THE COMMODORE 64
by Richard Allen King
250 pp., illustr. Ref. 231-0
This clear, straightforward guide will give you a firm grasp on Multiplan's function, as well as provide a collection of useful template programs.

WORD PROCESSING

PRACTICAL WORDSTAR USES
by Julie Anne Arca
303 pp., illustr. Ref. 107-1
Pick your most time-consuming office tasks and this book will show you how to streamline them with WordStar.

THE COMPLETE GUIDE TO MULTIMATE
by Carol Holcomb Dreger
250 pp., illustr. Ref. 229-9
A concise introduction to the many practical applications of this powerful word processing program.

THE THINKTANK BOOK
by Jonathan Kamin
200 pp., illustr., Ref. 224-8
Learn how the ThinkTank program can help you organize your thoughts, plans and activities.

PRACTICAL MULTIMATE USES
by Chris Gilbert
275 pp., illustr., Ref. 276-0
Includes an overview followed by practical business techniques, this covers documentation, formatting, tables, and Key Procedures.

MASTERING WORDSTAR ON THE IBM PC
by Arthur Naiman
200 pp., illustr., Ref. 250-7
The classic Introduction to WordStar is now specially presented for the IBM PC, complete with margin-flagged keys and other valuable quick-reference tools.

MASTERING MS WORD
by Mathew Holtz
365 pp., illustr., Ref. 285-X
This clearly-written guide to MS WORD begins by teaching fundamentals quickly and then putting them to use right away. Covers material useful to new and experienced word processors.

PRACTICAL TECHNIQUES IN MS WORD
by Alan R. Neibauer
300 pp., illustr., Ref. 316-3
This book expands into the full power of MS WORD, stressing techniques and procedures to streamline document preparation, including specialized uses such as financial documents and even graphics.

INTRODUCTION TO WORDSTAR 2000

by David Kolodnay and Thomas Blackadar

292 pp., illustr., Ref. 270-1

This book covers all the essential features of WordStar 2000 for both beginners and former WordStar users.

PRACTICAL TECHNIQUES IN WORDSTAR 2000

by John Donovan

250 pp., illustr., Ref. 272-8

Featuring WordStar 2000 Release 2, this book presents task-oriented tutorials that get to the heart of practical business solutions.

MASTERING THINKTANK ON THE 512K MACINTOSH

by Jonathan Kamin

264 pp., illustr., Ref. 305-8

Idea-processing at your fingertips: from basic to advanced applications, including answers to the technical question most frequently asked by users.

DATABASE MANAGEMENT SYSTEMS

UNDERSTANDING dBASE III PLUS

by Alan Simpson

415 pp., illustr., Ref. 349-X

Emphasizing the new PLUS features, this extensive volume gives the database terminology, program management, techniques, and applications. There are hints on file-handling, debugging, avoiding syntax errors.

UNDERSTANDING dBASE III

by Alan Simpson

250 pp., illustr., Ref. 267-1

The basics and more, for beginners and intermediate users of dBASEIII. This presents mailing label systems, bookkeeping and data management at your fingertips.

ADVANCED TECHNIQUES IN dBASE III

by Alan Simpson

505 pp., illustr., Ref. 282-5

Intermediate to experienced users are given the best database design techniques, the primary focus being the development of user-friendly, customized programs.

MASTERING dBASE III: A STRUCTURED APPROACH

by Carl Townsend

338 pp., illustr., Ref. 301-5

Emphasized throughout is the highly successful structured design technique for constructing reliable and flexible applications, from getting started to advanced techniques. A general ledger program is used as the primary illustration for the examples.

UNDERSTANDING dBASE II

by Alan Simpson

260 pp., illustr., Ref. 147-0

Learn programming techniques for mailing label systems, bookkeeping, and data management, as well as ways to interface dBASE II with other software systems.

ADVANCED TECHNIQUES IN dBASE II

by Alan Simpson

395 pp., illustr. Ref., 228-0

Learn to use dBASE II for accounts receivable, recording business income and expenses, keeping personal records and mailing lists, and much more.

Integrated Software

JAZZ ON THE MACINTOSH

by Joseph Caggiano and Michael McCarthy

431 pp., illustr., Ref. 265-5

Each chapter features as an example a business report which is built on throughout the book in the first section of each chapter. Chapters then go on to detail each application and special effects in depth.

SYBEX introduces one of the most sophisticated communications packages for your IBM-PC at the cost of a book!

Connect with **Elf**—

"as easy to use as it is powerful . . . easy to understand . . . simple enough for beginners . . . satisfies the needs of sophisticated users . . . becomes practically automatic . . . "

—PC Products

Elf automates virtually all calls and file transfers and turns your computer into a remote.

- Menu driven
- Easy to install
- Personality modules
- Crash-proof error handling
- Password protected post mode
- Supports XMODEM protocol
- Includes SYBEX-quality documentation

$49.95
(Diskette and Book)

ORDER NOW! CALL: 800-227-2346

SYBEX Computer Books are different.

Here is why . . .

At SYBEX, each book is designed with you in mind. Every manuscript is carefully selected and supervised by our editors, who are themselves computer experts. We publish the best authors, whose technical expertise is matched by an ability to write clearly and to communicate effectively. Programs are thoroughly tested for accuracy by our technical staff. Our computerized production department goes to great lengths to make sure that each book is well-designed.

In the pursuit of timeliness, SYBEX has achieved many publishing firsts. SYBEX was among the first to integrate personal computers used by authors and staff into the publishing process. SYBEX was the first to publish books on the CP/M operating system, microprocessor interfacing techniques, word processing, and many more topics.

Expertise in computers and dedication to the highest quality product have made SYBEX a world leader in computer book publishing. Translated into fourteen languages, SYBEX books have helped millions of people around the world to get the most from their computers. We hope we have helped you, too.

For a complete catalog of our publications:

SYBEX, Inc. 2021 Challenger Drive, #100, Alameda, CA 94501
Tel: (415) 523-8233/(800) 227-2346 Telex: 336311

SYBEX

THE MS-DOS HANDBOOK

The MS-DOS Handbook, Second Edition, is a manual for users and programmers of the large family of IBM compatibles and other computers running versions of MS-DOS, including Version 3. Operating systems expert Richard Allen King offers a detailed introduction to the inner workings of MS-DOS, lending the benefit of his experience to anyone who wants to master and use the tools of this popular system.

The MS-DOS Handbook is unique in providing two complete reference books in one:

The Programmer's Handbook—with detailed specifications for file and disk operations, keyboard, monitor and port control, memory management, and more. It includes descriptions of all operating system functions needed to write MS-DOS programs, with examples of how to use them; plus the author's own guidelines for effective programming.

The User's Handbook—designed to help users make the most of the system, including an introduction to EDLIN, DOS files, configuring the system, batch operations, monochrome and color screens, the serial port, and DEBUG.

You'll also find—

- a straightforward discussion of compatibility issues
- a guide to the different MS-DOS version numbers
- complete maps and tables for programmer's reference

New in the Second Edition:

- full details on the expanded capabilities of Version 3
- extended treatment of network operations
- expanded chapters on error reporting, configuring the system, and the system debugger
- a complete MS-DOS command summary

About the Author: Richard Allen King is an MS-DOS expert and computer consultant. He studied mathematics and engineering at London University. He has done extensive work in the development of operating systems, and has directed the production and installation of a wide variety of software products. Mr. King is also the author of *Doing Business with Multiplan* and *The IBM PC-DOS Handbook*. He lives in Berkeley, California.

SYBEX books bring you skills—not just information. As computer experts, educators, and publishing professionals, we care—and it shows. You can trust the SYBEX label of excellence.

ISBN 0-89588-352-X

U.S. $19.95